

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

«Доверенная разработка программных пакетов
и сборка программного обеспечения»

Москва 2024

УДК 004(075.8)

ББК 32.973.26-018.2

Учебно-методическое пособие
«Доверенная разработка программных пакетов
и сборка программного обеспечения»

Пособие применяется при обучении по следующим дисциплинам:

«Инфраструктура создания ПО для отечественных архитектур микропроцессоров»

«Сборка программного обеспечения в отечественных репозиториях»

«Разработка программного обеспечения для отечественных архитектур микропроцессоров»

Авторы:

М. А. Фоканова, М. О. Алексеева, А. А. Калинин, И. А. Мельников, Д. Н. Воропаев,
А. А. Лимачко, В. А. Синельников, В. А. Соколов, В. П. Капицин, А. В. Абрамов,
В. Л. Чёрный, В. С. Малиночкин

Под редакцией В. С. Малиночкина.

Благодарим сотрудников «Базальт СПО» за участие в разработке учебно-методического пособия.

Коллектив авторов выражает благодарность за поддержку разработки настоящего учебно-методического пособия Автономной некоммерческой организации «Координационный центр национального домена сети Интернет».

Издание утверждено советом факультета Кибернетики и информационной безопасности Московского технического университета связи и информатики.

Протокол №3 от 26 ноября 2024 года.

Рецензент: В.А. Кутуков, к.т.н.

ISBN 978-5-6050065-5-8

Оглавление

| | |
|--|----|
| Инфо | 2 |
| Введение | 5 |
| Глава 1. Пакетный менеджер | 7 |
| 1.1 RPM: основной пакетный менеджер в «Альт Платформа» | 9 |
| 1.2 АРТ: инструменты управления пакетами | 10 |
| 1.3 Вопросы для самопроверки | 14 |
| Глава 2. Основные команды пакетного менеджера RPM | 15 |
| 2.1 Установка RPM-пакета | 16 |
| 2.2 Проверка установки пакета в системе | 18 |
| 2.3 Просмотр файлов пакета, установленного в системе | 18 |
| 2.4 Просмотр недавно установленных пакетов | 19 |
| 2.5 Поиск пакета в системе | 19 |
| 2.6 Проверка файла, относящегося к пакету | 20 |
| 2.7 Вывод информации о пакете | 20 |
| 2.8 Обновление пакета | 21 |
| 2.9 Вопросы для самопроверки | 22 |
| Глава 3. Общая информация о сборке RPM-пакета | 23 |
| 3.1 Набор инструментов, необходимый для сборки | 24 |
| 3.2 Программное обеспечение для сборки RPM-пакетов | 25 |
| 3.3 Описание RPM-пакета | 26 |
| 3.4 Рабочее пространство для сборки RPM-пакетов | 27 |
| 3.5 Описание SPEC-файла | 28 |
| 3.6 Пример .spec-файла | 31 |
| 3.7 RPM макросы | 34 |
| 3.8 Вопросы для самопроверки | 36 |
| Глава 4. Инструмент rpmbuild | 38 |
| 4.1 Описание инструмента и сборка rpmbuild | 38 |
| 4.2 Вопросы для самопроверки | 40 |
| Глава 5. Инструмент Hasher | 41 |
| 5.1 Настройка Hasher | 41 |

| | | |
|--------------------------------------|---|-----------|
| 5.2 | Описание системы Hasher | 44 |
| 5.3 | Сборка в Hasher | 46 |
| 5.4 | Вопросы для самопроверки | 47 |
| Глава 6. Инструмент GEAR | | 48 |
| 6.1 | Описание GEAR | 49 |
| 6.2 | Правила экспорта | 49 |
| 6.3 | Основные типы устройства gear-репозитория | 51 |
| 6.4 | Работа с GEAR | 53 |
| 6.5 | Вопросы для самопроверки | 57 |
| Глава 7. Примеры и упражнения | | 58 |
| 7.1 | Базовые инструменты grm | 58 |
| 7.2 | Сборка в Hasher | 60 |
| 7.3 | Сборка с GEAR | 63 |
| 7.4 | Обновление ранее собранного пакета | 65 |
| Заключение | | 67 |

Введение

Учебно-методическое пособие «Доверенная разработка программных пакетов и сборка программного обеспечения» предназначено для очного и дистанционного обучения студентов по программам бакалавриата, специалитета, магистратуры и дополнительного профессионального образования базовой кафедры общественно-государственного объединения «Ассоциация документальной электросвязи» (АДЭ) «Технологии электронного обмена данными» (ТЭОД) в Московском техническом университете связи и информатики (МТУСИ).

Пособие состоит из введения, заключения, семи глав и содержит практикумы по следующим направлениям:

- Пакетный менеджер.
- Основные команды пакетного менеджера RPM.
- Общая информация о сборке rpm-пакета.
- Инструмент `rpmbuild`.
- Инструмент `Hasher`.
- Инструмент `GEAR`.
- Примеры и упражнения.

Пособие включает сведения о программной платформе лабораторного практикума на базе отечественных операционных систем семейства «Альт».

Семество «Альт» включает в себя линейку дистрибутивов разного назначения для различных аппаратных архитектур. В репозитории Sisyphus поддерживаются архитектуры: `i586`, `x86_64`, `armh` (`armv7`), `aarch64` (`armv8`), E2K (с третьего по шестое поколение), `riscv64`, `mipsel`, `loongarch`.

Дистрибутив — это составное произведение, в составе которого есть программа для дистрибуции (установки), называемая инсталлятор, и набор системного и прикладного ПО. В основе всех дистрибутивов лежат пакеты свободного программного обеспечения.

Свободное программное обеспечение (СПО) — это программное обеспечение, распространяемое на условиях простой (неисключительной) лицензии, которая позволяет пользователю:

1. использовать программу для ЭВМ в любых, не запрещённых законом, целях;
2. получать доступ к исходным текстам (кодам) программы как в целях изучения и адаптации, так и в целях переработки программы для ЭВМ; распространять программу (бесплатно или за плату, по своему усмотрению);
3. вносить изменения в программу для ЭВМ (перерабатывать) и распространять экземпляры изменённой (переработанной) программы с учётом возможных требований наследования лицензии;
4. в отдельных случаях (copyleft лицензия) распространять модифицированную компьютерную программу пользователем на условиях, идентичных тем, на которых ему предоставлена исходная программа.

Примерами свободных лицензий являются:

1. **GNU General Public License**¹. Version 3, 29 June 2007 (Стандартная общественная лицензия GNU. Версия 3, от 29 июня 2007 г.).
2. **BSD license**² — New Berkley Software Distribution license (Модифицированная программная лицензия университета Беркли).

СПО отлично подходит для целей обучения и для разработки собственных решений, потому что весь код доступен для изучения и модификации. Однако авторы настоятельно советуют всем, кто использует СПО для построения своих программных продуктов, учитывать особенности лицензирования не только самих пакетов, но и входящих в их состав библиотек. Если вы используете copyleft библиотеку, это обязывает вас распространять свою программу под аналогичной лицензией, т.е. любой человек, который получит вашу программу легальным способом, может потребовать предъявить ему исходный код.

¹<https://www.gnu.org/licenses/gpl-3.0.html>

²https://en.wikipedia.org/wiki/BSD_licenses

Глава 1

Пакетный менеджер

Операционная система состоит из разнообразных компонентов: программ, библиотек, скриптов и приложений. Число компонентов может достигать тысячи единиц, в каждой из которых могут быть включены десятки файлов. Для удобства работы пользователя системные компоненты в **Linux** представлены в виде пакетов¹. Пакет объединяет в общий архив сходные по назначению файлы: исполняемые программы, наборы библиотек, скрипты или конфигурационные файлы и данные. Пользователь выбирает программы, ориентируясь на общеизвестное имя, устанавливает, обновляет, проверяет, удаляет их, не вдаваясь в отдельные детали подбора всех необходимых файлов и компонентов. Работа с пакетами позволяет сохранять целостность программы со всеми её компонентами.

Пакет — это специально подготовленный архив, содержащий файлы данных, конфигурационные файлы, управляющую информацию и метаданные. Метаданные пакета содержат полное имя, номер версии, принадлежность архитектуре, цифровую подпись, описание пакета, информацию о лицензии и некоторую служебную информацию о сборке. Управляющая информация пакета содержит сценарии установки и удаления пакета, зависимости устанавливаемого пакета от других пакетов, краткое описание и прочую информацию, которую использует менеджер пакетов. Пакеты принято хранить в специальном хранилище — **репозитории пакетов**.

Для удобства работы команды разработчиков придумали собственные форматы архивов:

- **RPM (.rpm)**. Разработан компанией **Red Hat**. Применяется в системе «Альт», Ред ОС, RHEL и CentOS.
- **DEB (.deb)**. Формат пакетов дистрибутива **Debian**, а также **Ubuntu**.

¹Курячий Г., Маслинский К. (2010). Операционная система Linux. Курс лекций. ДМК Пресс.

- **TAR.XZ** (`.tar.xz`). Применяется в дистрибутивах ArchLinux и Manjaro.
- **APK** (`.apk`). Применяется в операционной системе Android.

Каждый пакет определяется именем, архитектурой системы, под которую он собран, номером её версии и номером релиза этой программы в дистрибутиве. Если пакет не зависит от архитектуры процессора, то в качестве архитектуры указывается «noarch».

Например, `admc-0.17.0-alt1.e2kv6.rpm`:

Имя: `admc`
Номер версии: `0.17.0`
Номер релиза: `alt1`
Архитектура: `e2kv6`

Зависимость пакета — потребность компонентов в составе пакета в ресурсах или компонентах прочих пакетов. Может случиться, что для успешного запуска программы из одного пакета необходимы библиотеки или другие ресурсы, которые находятся в другом пакете. В таком случае говорят о зависимости пакета от одного или нескольких пакетов. Пакетный менеджер запретит установку пакета в систему без установки всех необходимых пакетов, удовлетворяющих зависимости.

Пакетный менеджер (система управления пакетами) — это система управления: установкой, удалением, настройкой и обновлением пакетов. Пакетные менеджеры средствами входящих в их состав утилит упрощают для пользователя процесс управления пакетами в операционной системе. Пакетный менеджер ведёт учёт пакетов, установленных в системе. Существует менеджер зависимостей — специальная программа, подбирающая пакеты, зависимые друг от друга, и загружающая эти пакеты из хранилища². Менеджер зависимостей подбирает правильные версии пакетов и определяет порядок их установки. При помощи менеджера зависимостей можно узнать с каким пакетом поставляется тот или иной файл.

Задачи пакетного менеджера:

- **установка программ.** Позволяет устанавливать программы из центрального хранилища или из локальных источников;
- **обновление программ.** Позволяет обновлять установленные программы до последних версий, представленных в хранилище;
- **удаление программ.** Позволяет безопасно удалять программы и все связанные с ними файлы;
- **управление зависимостями.** Автоматически устанавливает и управляет зависимостями программ;

²Кетов Д. (2021). Внутреннее устройство Linux. 2-е изд., перераб. и доп. БХВ-Петербург.

- **проверка целостности пакетов.** Предотвращает конфликты при установке новых программ, обеспечивая целостность системы.

Утилиты пакетного менеджера позволяют:

- узнать информацию о пакете;
- определить пакет, которому принадлежит установленная программа;
- определить список компонентов, установленных из указанного пакета.

Среди утилит пакетного менеджера можно выделить две категории — низкоуровневые и высокоуровневые.

- **Низкоуровневые утилиты пакетного менеджера.** Используются для установки локальных пакетов, загруженных вручную пользователем или высокоуровневым пакетным менеджером.
- **Высокоуровневые утилиты.** Применяются для поиска и скачивания пакетов из репозиториев. В процессе работы могут задействовать низкоуровневые менеджеры для установки загруженных программ.

В операционной системе «Альт» используется формат пакетов `.rpm`. Пакеты `rpm` хранятся в удалённом хранилище. Для работы с такими пакетами применяется низкоуровневый пакетный менеджер RPM и консольные утилиты APT (Advanced Packaging Tool)³.

- **RPM** используется для просмотра, сборки, установки, инспекции, проверки, обновления и удаления отдельных программных пакетов. Каждый такой пакет состоит из набора файлов и информации о пакете, включающей название, версию, описание пакета и т. д.
- **APT** умеет автоматически разрешать зависимости при установке, обеспечивает установку из нескольких источников и целый ряд других уникальных возможностей, включая получение последней версии списка пакетов из репозитория и обновление системы.

1.1 RPM: основной пакетный менеджер в «Альт Платформа»

В дистрибутивах «Альт» применяется пакетный менеджер RPM. RPM Package Manager — это семейство пакетных менеджеров, применяемых в различных дистрибутивах GNU/Linux. Практически каждый крупный проект, использующий RPM, имеет свою версию пакетного менеджера, отличающуюся от остальных.

Различия между представителями семейства RPM выражаются в:

³<https://wiki.altlinux.ru/QuickStart/PkgManagment>

- наборе макросов, используемых в `.spec`-файлах;
- различии сборки `rpm`-пакетов «по умолчанию» — при отсутствии каких-либо указаний в `.spec`-файлах, формате строк зависимостей;
- отличиях в семантике операций (например, в операциях сравнения версий пакетов);
- отличиях в формате файлов.

Обратим внимание на то, что в операционных системах ALT ведется самостоятельная разработка формата `.rpm` и пакетного менеджера `APT`. Набор утилит `APT` в ALT отличается от аналогичной по названию программы в Debian, также как и `RPM` отличается от аналогичного пакетного менеджера в RedHat.

1.2 APT: инструменты управления пакетами

`APT` — часть системы управления пакетами в дистрибутивах «Альт». **Advanced Packaging Tool** (усовершенствованный инструмент работы с пакетами) это набор утилит, позволяющий управлять пакетами. `APT` поддерживает загрузку пакетов из хранилища (репозитория).

Хранилище(репозиторий) — в общем виде хранилище данных.

В операционной системе «Альт» пакетный менеджер работает с репозиторием `rpm`-пакетов.

Репозиторий пакетов — это замкнутая совокупность компонентов системы с поддерживаемой целостностью и метаинформацией о них, то есть структурированные компоненты с формализованными инструкциями по установке и разрешенными зависимостями.

Хранилище состоит из двух частей — индексы (списки пакетов со служебной информацией) и хранилище (структурированные файлы пакетов). `APT` в зависимости от настроек может использовать удалённый репозиторий с помощью сетевого протокола (например, `ftp`) или локальный репозиторий (например, на оптическом диске). Список источников пакетов хранится в файле `/etc/apt/sources.list` и в каталоге `/etc/apt/sources.list.d/`. В системе «Альт» применяется графическая оболочка для `APT` — программа `Synaptic`⁴. Утилита `apt-get` значительно упрощает процесс установки программ в командном режиме.

⁴APT и Synaptic развиваются ALT Linux Team, не нужно сравнивать реализации с аналогичными утилитами в Debian

Для сокращения команд, встречающихся в тексте, используется нотация:



- команды **без административных привилегий** начинаются с символа «\$»;
- команды **с административными привилегиями** начинаются с символа «#».

Команда `apt-get` выведет описание и возможности утилиты `apt-get`:

```
$ apt-get
apt 0.5.15lorg2 для linux e2k собран May 22 2024 13:22:49
Использование: apt-get [параметры] команда
    apt-get [параметры] install|remove пакет1 [пакет2 ...]
    apt-get [параметры] source пакет1 [пакет2 ...]
```

`apt-get` предоставляет простой командный интерфейс для получения и установки пакетов. Чаще других используются команды `update` (обновить) и `install` (установить).

Команды:

- `update` — получить обновлённые списки пакетов;
- `upgrade` — произвести обновление системы;
- `install` — установить новые пакеты;
- `remove` — удалить пакеты;
- `source` — скачать архивы исходников;
- `build-dep` — установить всё необходимое для сборки исходных пакетов;
- `dist-upgrade` — обновление системы в целом;
- `clean` — удалить скачанные ранее архивные файлы;
- `autoclean` — удалить давно скачанные архивные файлы;
- `check` — удостовериться в отсутствии неудовлетворённых зависимостей;
- `dedup` — удаление неразрешенных дубликатов пакетов.

Параметры:

- `-h` — краткая справка;
- `-q` — скрыть индикатор процесса;
- `-qq` — не показывать ничего кроме сообщений об ошибках;

- `-d` — получить пакеты и выйти БЕЗ их установки или распаковки;
- `-s` — симулировать упорядочение вместо реального исполнения;
- `-y` — автоматически отвечать «ДА» на все вопросы;
- `-f` — пытаться исправить положение, если найдены неудовлетворённые зависимости;
- `-m` — пытаться продолжить, если часть архивов недоступна;
- `-u` — показать список обновляемых пакетов;
- `-b` — собрать пакет после получения его исходника;
- `-D` — при удалении пакета стремиться удалить компоненты, от которых он зависит;
- `-V` — подробно показывать номера версий;
- `-c=?` — использовать указанный файл конфигурации;
- `-o=?` — изменить любой из параметров настройки (например: `-o dir::cache=/tmp`).

Более полное описание доступно на страницах руководства `man: apt-get`, `sources.list` и `apt.conf`:

- `$ man apt-get`
- `$ man sources.list`
- `$ man apt.conf`

В ОС «Альт» утилита `apt-get` использует основной пакетный менеджер RPM Package Manager для установки, обновления, удаления пакетов, управления зависимостями. Обе утилиты `rpm` и `apt-get` позволяют установить, обновить или удалить пакет.

Отличия `rpm` и `apt-get`:

- `apt-get` учитывает зависимости устанавливаемого пакета;
- `apt-get` умеет работать с репозиторием в целом:
 - искать пакеты;
 - вычислять список обновлений — находить разницу версий пакетов, установленных локально и хранящихся в репозитории.
- `apt-get` получает информацию из пакетов, используя `rpm`.

Утилита **rpm** подразумевает работу с конкретными пакетами. Пользователь самостоятельно принимает решения, связанные с зависимостями пакетов при работе с **RPM**. Утилита **apt-get** вычисляет и устанавливает необходимые пакеты из репозитория, чтобы удовлетворить зависимости для каждого **rpm**-пакета. Утилита **apt-get** самостоятельно не устанавливает пакеты, а использует для этого **rpm**.



Установка пакетов в «Альт Платформа» осуществляется с помощью утилиты **apt**

Целостность компонентов репозитория пакетов обеспечивает инфраструктура разработки операционной системы «Альт». Результат работы инфраструктуры это репозиторий пакетов. Каждый пакет репозитория формируется на основе исходных данных пакета. Множество таких исходных данных для каждого пакета составляют **git-репозиторий**.

Git-репозиторий — хранилище исходных данных с сохранением истории изменений каждого файла хранилища. В данном контексте мы подразумеваем множество репозиториев исходных данных компонентов системы (будь то ядро операционной системы, служебная библиотека, текстовый редактор, сервер для обслуживания электронных сообщений или набор изображений для оформления графической среды), входящих в операционную систему **ALT**.

Инфраструктура разработки **ALT** на основе подготовленных для сборки в пакеты исходных данных компонентов системы выполняет типовые операции:

- собирает компонент в соответствии с подготовленными инструкциями;
- проверяет целостность каждого компонента;
- проверяет зависимости и целостность связанных компонентов;
- добавляет пакет в репозиторий пакетов, если все условия выполнены.

Поддерживаемые в **ALT** репозитории пакеты называются «Альт Платформа» и обладают уникальным идентификатором репозитория. В декабре 2023 года сформирован и поддерживается репозиторий **p10** под названием «Альт Платформа 10».

Репозиторий пакетов и утилиты **APT** вместе автоматизируют процессы управления установкой, обновления и удаления программного обеспечения, исключают риск случайного повреждения целостности операционной системы и прикладных программ.

Процесс взаимодействия пользователя с **APT**:

- средствами **APT** по запросу пользователя загружаются метаданные из репозитория;

- АРТ получает от пользователя информацию о том, какие именно пакеты обновить или установить;
- АРТ проверяет зависимости и возможные конфликты компонентов;
- АРТ предлагает пути решения, например, загрузку новых пакетов из репозитория, установку дополнительных или обновление имеющихся пакетов.



Для обновления практически всего программного обеспечения (за исключением ядра операционной системы) на локальном компьютере до новой версии необходимо выполнить команды:

```
# apt-get update
# apt-get dist-upgrade
```

При использовании АРТ и обновляемого стабильного репозитория операционная система может служить на компьютере годами, гарантировано обновляясь до новых версий.

1.3 Вопросы для самопроверки

1. Что такое пакет?
2. Какие форматы пакетов вы знаете?
3. Что такое зависимость пакета?
4. Что такое репозиторий пакетов?
5. Какие низкоуровневые пакетные менеджеры вы знаете?
6. Какой пакетный менеджер используется в «Альт Платформа»?
7. Вы обнаружили в системе пакет `nagios-domain-discovery-0.1.1-alt1.noarch.rpm`, определите его имя, версию, релиз и архитектуру.
8. Верно ли утверждение, что утилита `apt-get` при установке пакетов требует явного указания всех зависимых пакетов?
9. Для чего используется команда `apt-get update`?
10. Выполнится ли успешно команда `$ apt-get dist-upgrade`?

Глава 2

Основные команды пакетного менеджера RPM

Управлять пакетами можно из командной строки при помощи программы `rpm`, которая имеет следующий синтаксис:

```
rpm [параметры]
```

Пакетный менеджер RPM предоставляет базовые возможности для управления пакетами. Основной набор команд¹ позволяет установить, удалить, обновить пакеты, получить разнообразную информацию о самих пакетах и их содержимом:

- **Проверка установки пакета в системе:** `rpm -q ИМЯ_ПАКЕТА`
Эта команда проверяет установлен ли пакет в системе. `rpm -q` означает `query` (запрос). Используется с дополнительными ключами для выполнения запросов различного назначения.
- **Информация о пакете:** `rpm -qi ИМЯ_ПАКЕТА`
`rpm -qi` выводит подробную информацию о конкретном установленном пакете. `rpm -qi` означает «`query information`» (запросить информацию).
- **Просмотр установленных пакетов:** `rpm -qa`
Эта команда выводит список всех установленных пакетов в системе. `-a: --all` (все). `rpm -qa` означает «`query all`» (запрос всего).
- **Проверка зависимостей пакета:** `rpm -qR ИМЯ_ПАКЕТА`
`rpm -qR` выводит список зависимостей (другие пакеты), необходимых для работы указанного пакета. `-R: -requires` (нуждается). `rpm -qR` означает «`query requires`» (запрос зависимостей).

¹https://wiki.altlinux.ru/Команды_RPM

- **Проверка файла на принадлежность пакету:** `rpm -qf ФАЙЛ`
Команда `rpm -qf` определяет к какому пакету принадлежит указанный файл. `-f: --file (файл)`. `rpm -qf` означает «query file» (запрос файла).
- **Просмотр файлов пакета:** `rpm -ql ИМЯ_ПАКЕТА`
`rpm -ql` выводит список всех файлов, содержащихся в установленном пакете. `-l: --list (список)`. `rpm -ql` означает «query list» (запрос списка).
- **Установка пакета:** `rpm -i ФАЙЛ_ПАКЕТА`
Команда `rpm -i` используется для установки пакета из файла `.rpm`. `-i: --install (установить)`. Например, `rpm -i package.rpm` установит содержимое пакета `package` в систему.
- **Удаление пакета:** `rpm -e ИМЯ_ПАКЕТА`
`rpm -e` удаляет установленный пакет. `-e: --erase (стереть)`. Например, `rpm -e package` удалит пакет с именем `package`.
- **Обновление пакета:** `rpm -U ФАЙЛ_ПАКЕТА` или `rpm -F ФАЙЛ_ПАКЕТА`
Команда `rpm -U` обновит пакет до новой версии или установит его, `rpm -F` только обновит пакет, если он уже установлен. `-U: --upgrade (обновить)`, `-F: --freshen (освежить)`.
- **Проверка целостности пакета:** `rpm -V ИМЯ_ПАКЕТА` или `ФАЙЛ_ПАКЕТА`
`rpm -V` проверяет целостность файлов в пакете, сравнивая их с информацией в базе данных `rpm`. `-V: --verify (проверить)`.

Дополнительные ключи:

`-v: --verbose` (подробно). Подробный вывод, если доступно. Например, `rpm -iv` означает «install verbose» (установка подробно) и используется для вывода более подробной информации в процессе установки пакета. Подробный вывод существует не для всех ключей утилиты.

`--quiet`: (тихо). Вывести минимальный набор сообщений, если возможно. Например, `rpm -i --quiet` означает «install quiet» (установка тихая).



Справку по ключам можно получить, набрав в консоли команду `rpm --help`.

2.1 Установка RPM-пакета



В команде должен быть указан файл пакета или полный путь к нему.

Для установки пакета из rpm-файла используйте команду:

```
# rpm -i ФАЙЛ_ПАКЕТА
```

Синтаксис команды:

```
# rpm {-i | --install} [опции-установки] ФАЙЛ_ПАКЕТА
```



Для работы с командой потребуются права суперпользователя. Их можно получить через команду `su-` либо команду `sudo`

Пример выполнения команды:

```
# rpm -i gpupdate-0.9.12.6-alt1.src.rpm
```

В конце команды возможно указать дополнительные опции:

- `--nodeps` — не проверять зависимости пакета;
- `--replacepkgs` или `--reinstall` — переустановить пакет.

Подробный вывод. Для отображения прогресса установки используйте дополнительные параметры `-v` и `-h`.

- `-v` — вывести детальные сообщения;
- `-h` — вывести «#» строку индикатора прогресса по мере установки пакета (используется с `-v`).

Пример выполнения команды:

```
# rpm -ivh gpupdate-0.9.12.6-alt1.src.rpm
```

```
Подготовка...
#####[100%]
```

```
Обновление / установка...
```

```
1: gpupdate-0.9.12.6-alt1.src.rpm
#####[100%]
```


2.2 Проверка установки пакета в системе

Чтобы проверить, установлен ли пакет, введите следующую команду:

```
$ rpm -q ИМЯ_ПАКЕТА
```

Пример:

```
$ rpm -q gupupdate
      gupupdate-0.9.12.2-alt2.noarch

$ rpm -q mediinfo
      пакет mediinfo не установлен
```

2.3 Просмотр файлов пакета, установленного в системе

Чтобы получить список файлов пакета, введите следующую команду:

```
$ rpm -ql ИМЯ_ПАКЕТА
```

В этой команде используется ключ «-l» (list).



Для развёрнутой информации укажите ключ «-i».

Пример использования:

```
$ rpm -ql admc
      /usr/bin/admc
      /usr/lib64/libadldap.so
      /usr/share/applications/admc.desktop
      /usr/share/doc/admc-0.17.0
      /usr/share/doc/admc-0.17.0/CHANGELOG.txt
      /usr/share/doc/admc-0.17.0/CHANGELOG_ru.txt
      /usr/share/doc/admc-0.17.0/README.md
      /usr/share/icons/hicolor/scalable/apps/admc.svg
      /usr/share/man/man1/admc.1.xz
```

Чтобы узнать содержимое неустановленного rpm-пакета, используйте команду:

```
$ rpm -qlp ИМЯ_ПАКЕТА
```

Пример выполнения команды:

```
$ rpm -qlp udisks2-2.9.4-alt1.1.src.rpm \
      udisks2-2.9.4.tar.bz2 udisks2.control udisks2.spec
```


2.4 Просмотр недавно установленных пакетов

Чтобы получить список последних установленных пакетов, введите следующую команду:

```
$ rpm -qa --last | head
```

Вывод:

| | |
|--|-------------------------|
| wireshark-qt5-4.2.7-alt1.e2kv6 | Пн 14 окт 2024 13:57:14 |
| wireshark-base-4.2.7-alt1.e2kv6 | Пн 14 окт 2024 13:57:13 |
| snmp-mibs-std-0.3-alt3.noarch | Пн 14 окт 2024 13:57:06 |
| qt6-5compat-common-6.6.2-alt1.noarch | Пн 14 окт 2024 13:57:06 |
| libsmi-0.5.0-alt2.e2kv5 | Пн 14 окт 2024 13:57:06 |
| libqt6-printsupport-6.6.2-alt4.E2K.1.e2kv6 | Пн 14 окт 2024 13:57:06 |
| libqt6-core5compat-6.6.2-alt1.e2kv6 | Пн 14 окт 2024 13:57:06 |
| libmaxminddb-1.9.1-alt1.e2kv6 | Пн 14 окт 2024 13:57:06 |
| liblua5.1-preinstall-5.1.5-alt21.e2kv6 | Пн 14 окт 2024 13:57:06 |
| liblua5.1-5.1.5-alt21.e2kv6 | Пн 14 окт 2024 13:57:06 |

Команда `rpm -qa --last` используется для вывода списка всех установленных пакетов, отсортированных по времени их установки. Пакеты будут отсортированы в порядке убывания времени установки — самые последние установленные пакеты отобразятся в верхней части списка.

Фильтрация вывода: утилита `grep` отфильтрует вывод и поможет найти искомый пакет. Например, следующая команда выведет информацию только о тех пакетах, название которых содержит «`kernel`»:

```
$ rpm -qa --last | grep kernel
```

2.5 Поиск пакета в системе

Чтобы найти в системе необходимый пакет среди уже установленных, используйте утилиту `grep`. Утилита `grep` находит строки по запросу.

```
$ rpm -qa | grep ИМЯ_ПАКЕТА
```

Например, запрос:

```
$ rpm -qa | grep wireshark
wireshark-qt5-4.2.7-alt1.e2kv6
wireshark-base-4.2.7-alt1.e2kv6
```


2.6 Проверка файла, относящегося к пакету

Чтобы определить, какому пакету принадлежит указанный файл, используйте команду:

```
$ rpm -qf ФАЙЛ
```

Например, запрос:

```
$ rpm -qf /usr/share/wireshark/cfilters
```

Предоставит вывод:

```
wireshark-base-4.2.7-alt1.e2kv6
```

2.7 Вывод информации о пакете

Чтобы получить вывод подробной информации о конкретном установленном пакете² — название, версию и прочее — используйте команду:

```
$ rpm -qi ИМЯ_ПАКЕТА
```



В запросе указывается имя пакета из списка уже установленных в системе, либо путь к загруженному пакету.



Если в системе установлена одна версия пакета, можно указывать только имя. Если установлено больше одной версии, необходимо указывать конкретную версию и релиз(**имя-версия-релиз**).

Пример выполнения команды для установленного пакета в системе:

```
$ rpm -qi wireshark-qt5
```

```
Name       : wireshark-qt5
Version    : 4.2.7
Release    : alt1
Architecture: e2kv6
Install Date: Пн 14 окт 2024 13:57:14
Group      : Monitoring
Size       : 34633833
License    : GPLv2
Signature  : RSA/SHA1, C6 28 сен 2024 01:59:25, Key ID cea7f56e5689c9f0
Source RPM : wireshark-4.2.7-alt1.src.rpm
Build Date : C6 28 сен 2024 01:58:32
```

²https://www.inp.nsk.su/~bolkhov/teach/inpunix/make_rpm.ru.html


```
Build Host : mike-sisyphus_e2k.hasher.altlinux.org
Relocations : (not relocatable)
Packager   : Anton Farygin <rider@altlinux.org>
Vendor     : ALT Linux Team
URL        : http://www.wireshark.org/
Summary    : QT5 GUI for Wireshark package
Description :
This package contains QT5 GUI ie. the wireshark - application.
```

Пример выполнения команды для rpm-пакета, расположенного на диске, но не установленного в системе:

```
$ rpm -qip /путь/к/пакету/имя_пакета-версия-архитектура.rpm
```

Пример выполнения:

```
$ rpm -qip hasher_1.7.4-alt1_noarch.rpm

Name       : hasher
Version    : 1.7.4
Release    : alt1
Architecture: noarch
Install Date: (not installed)
Group      : Development/Other
Size       : 149877
License    : GPLv2+
Signature  : RSA/SHA1, Cp 22 мая 2024 22:22:48, Key ID cea7f56e5689c9f0
Source RPM : hasher-1.7.4-alt1.src.rpm
Build Date : Cp 22 мая 2024 22:22:44
Build Host : mike-sisyphus_e2k.hasher.altlinux.org
Relocations : (not relocatable)
Packager    : Arseny Maslennikov <arseny@altlinux.org>
Vendor      : ALT Linux Team
URL         : http://en.altlinux.org/Hasher
Summary     : Modern safe package building technology
Description :
Hasher is a set of tools for constructing chroot and safe building of
packages in the clean environment. It makes clean environment on every
new build. Hasher obtains packages from APT repositories so fast
network connection or local mirror is highly recommended.
```

2.8 Обновление пакета

Чтобы обновить пакет до новой версии, используйте команду:

```
# rpm -U ФАЙЛ_ПАКЕТА
```


Для обновления пакетов в пакетном менеджере RPM используют запрос с двумя типами ключей³:

1. `-U` — может устанавливать и обновлять пакеты;
2. `-F` — только обновляет уже установленные пакеты.

```
# rpm {-U | --upgrade} [опции-установки] ФАЙЛ_ПАКЕТА
```

Команда `rpm -U` (`Upgrade`) обновляет пакеты до новых версий. Если установлена старая версия пакета, `rpm -U` заменит старую версию на новую, обновив пакет. Если пакет не установлен, команда `rpm -U` установит его.

```
# rpm {-F | --freshen} [опции-установки] ФАЙЛ_ПАКЕТА
```

Команда `rpm -F` (`Freshen`) обновляет только те пакеты, которые уже установлены в системе. Если установлена старая версия пакета, `rpm -F` заменит старую версию на новую, обновив пакет. Если пакет не установлен, команда `rpm -F` ничего не сделает с этим пакетом⁴.

Пример обновления пакета:

```
# rpm -U admc-0.17.0-alt1-e2kv6.rpm
```

2.9 Вопросы для самопроверки

1. С помощью какого ключа к `rpm` можно получить информацию об установленном `rpm`-пакете?
2. Как получить информацию о `rpm`-пакете, если он ещё не установлен в системе, но у вас есть файл пакета?
3. Как получить список всех файлов `rpm`-пакета?
4. Верно ли что команда `rpm -Uvh` обновит пакет, а если его нет, то установит его?
5. Какой ключ отвечает за обновление/установку?
6. Как проверить установлен ли пакет с именем `foo` в вашей системе?
7. У вас произошёл сбой и, возможно, пострадала файловая система. Как проверить: все ли пакеты по прежнему консистентны?
8. Что сделает команда `$ rpm -qi mate-text-editor | grep License`?
9. Составьте команду, чтобы узнать есть ли у пакета подпись и какая.
10. Можно ли (пере)подписать `rpm`-пакет после сборки?

³<https://wiki.altlinux.ru/QuickStart/PkgManagment>

⁴<https://access.redhat.com/solutions/1189>

Глава 3

Общая информация о сборке RPM-пакета

Сборка — формирование пакета на основе специальных сборочных (spec) инструкций.

Сборка также устоявшееся определение компиляции исходного кода (формирование машинных инструкций), в случае RPM термин Сборка более общий, и в некоторых случаях включает компиляцию.

Использование специализированных пакетов, в отличие от обычного файлового архива, имеет ряд преимуществ¹:

- Пользователи могут использовать средства управления пакетами — пакетные менеджеры (например, Yum или PackageKit) для установки, переустановки, удаления, обновления и проверки rpm-пакетов.
- Пакетный менеджер RPM предполагает наличие базы данных, которая с помощью специализированных утилит позволяет получать информацию о пакетах в системе.
- Каждый пакет rpm содержит метаданные, описывающие компоненты пакета: версию, выпуск, размер, URL проекта, установочные инструкции и т. д.
- RPM позволяет брать оригинальные источники исходных данных и упаковывать их в пакеты с исходными данными (.src.rpm) и в бинарные пакеты (.rpm). В пакетах .src.rpm хранятся оригинальные исходные данные вместе со всеми изменениями (*.patch), а также сборочные инструкции (.spec) и дополнительная информация. В бинарных пакетах вместо исходного кода упакованы подготовленные файлы и скрипты установки, но нет сборочных инструкций. Ещё существуют случаи распространения пакетов без исходного кода и бинарных данных. В таких пакетах присутствуют скрипты для скачивания и модификации файлов, необходимые для работы приложения, распространяемого таким способом.

¹<https://rpm-packaging-guide-ru.github.io/#Why-Package-Software-with-RPM>

- Для обеспечения верификации подлинности **rpm**-пакетов используется механизм электронных цифровых подписей **GPG**. Он позволяет подписать **rpm**-пакет или обновить цифровую подпись: `rpm -addsign package.rpm` и `rpm -resign package.rpm`.
- Вы можете добавить свой пакет в **rpm**-репозиторий, что позволит клиентам легко находить и устанавливать ваше программное обеспечение.



Надо понимать, что процесс сборки пакета поддерживает предварительную подготовку исходных данных к использованию в операционной системе (например, такую как компиляция), но это не обязательно. Случай сборки уже готовых данных (например, графические изображения) иногда называют — упаковка в **rpm**-пакет.

Задача сборки пакета начинается со сбора всех необходимых компонентов и завершается этапами сборки и тестирования.

Классическая сборка **rpm**-пакетов состоит из следующих этапов:

- поиск исходных данных;
- написание инструкций сборки;
- сборка пакета.

3.1 Набор инструментов, необходимый для сборки

Опишем основные инструменты для сборки пакетов. Технологическую базу репозитория **Sisyphus** составляют адаптированные к нуждам команды разработчиков программы и специально разработанные решения²:

- **RPM** — в контексте данной темы рассматривается не только как менеджер пакетов, но и как набор инструментов для их сборки (**rpmbuild**). Отличительные особенности **RPM** в **Sisyphus** — удобное поведение «по умолчанию» для уменьшения количества шаблонного кода в **.спес**-файлах, обширный модульный набор макросов для упаковки различных типов пакетов, развитые механизмы автоматического вычисления межпакетных зависимостей при сборке пакетов, поддержка **set**-версий в зависимостях на разделяемые библиотеки, автоматическое создание пакетов с отладочной информацией с поддержкой зависимостей между такими пакетами.
- **Hasher** — инструмент, который позволяет производить сборку **rpm**-пакетов в изолированной среде, что обеспечивает повышенную безопасность и идентичный результат, не зависящий от стороннего программного обеспечения, установленного в системе.

²https://www.altlinux.org/Репозиторий_СПО#АПТ_и_репозиторий_пакетов.

- **GEAR** — инструмент, направленный на упрощение работы с источниками исходных данных для пакетов и облегчение процесса сборки.

3.2 Программное обеспечение для сборки RPM-пакетов

Перечислим набор базовых пакетов и входящих в их состав программ, необходимых для сборки rpm-пакета³. Необходимо отметить, что перечень может отличаться для различных исходных данных и выбранных инструментов сборки пакетов, которые подробнее будут разобраны позже.

- **rpmdevtools** — пакет с набором программ для сборки пакетов:
 - **rpmdev-setuptree** — утилита для создания структуры рабочих каталогов;
 - **rpmdev-newspec** — утилита для создания **spec**-файла.
- **rpmspec** — утилита работы с файлами спецификации — текстовыми файлами с расширением **.spec**. Утилита служит для проверки подготовленного **spec**-файла;
- **rpm-build** — пакет содержит сценарии и исполняемые программы, которые используются для сборки пакетов с помощью **RPM**:
 - **rpmbuild** — утилита сборки rpm-пакета из набора подготовленных файлов.
- **rpmlint** — утилита для тестирования собранного rpm-пакета;
- **rpm-utils** — пакет с набором программ для работы с rpm-пакетами;
- **gcc** — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU;
- **make** — инструмент GNU, упрощающий процесс сборки для пользователей;
- **python** — интерпретируемый интерактивный объектно-ориентированный язык программирования;
- **patch** — программа исправлений, которая применяет патчи к оригиналам;
- **gear** — пакет, содержащий утилиты для сборки пакетов rpm из gear-репозитория и управления gear-репозиториями;
- **hasher** — современная технология создания независимых от сборочной системы пакетов.

³https://www.altlinux.org/Сборка_пакета_с_нуля

3.3 Описание RPM-пакета

rpm-пакет — это специальный архив с файлами. Сам файл пакета состоит из четырёх секций: начального идентификатора, сигнатуры, бинарного заголовка и `srcio`-архива с файлами проекта в структуре каталогов⁴.

rpm-пакеты делятся на несколько категорий: пакеты с исходным кодом и бинарные пакеты.

- **rpm-пакет** (бинарный) — это архив с расширением `.rpm`. Такой пакет готов к установке в операционной системе средствами RPM.
- **srpm-пакет** (source RPM, пакет с исходным кодом) — это архив с расширением `.src.rpm`. SRPM содержит исходный код, патчи, если необходимо, и `spec`-файл. Эти пакеты содержат всю информацию для сборки пакета.

По инструкциям из `spec`-файла собирается бинарный rpm-пакет⁵. Инструкции содержат также информацию о правах доступа и их применении в процессе установки, скрипты, запускаемые при установке или удалении пакета.

Файлы бинарного пакета могут быть скомпилированы под определённую процессорную архитектуру (аппаратную платформу). На системах с разной процессорной архитектурой не получится использовать один и тот же собранный бинарный rpm-пакет без дополнительных настроек и манипуляций. Имя файла таких пакетов обычно содержит общепринятую маркировку архитектуры и имеет вид: `*.{архитектура}.rpm`

Существуют бинарные пакеты, которые не зависят от процессорной архитектуры. Такие пакеты содержат в своём имени маркировку архитектуры **noarch**. Имя платформо-независимого бинарного пакета заканчивается на: `.noarch.rpm`. Например, упакованные картинки рабочего стола или наборы иконок обычно имеют такое окончание.

На основе установленных в систему бинарных пакетах строится база данных в `/var/lib/rpm`. Вся информация о пакетах хранится в базе данных `Packages`.

Принято называть пакеты RPM:

`{имя-пакета}.{имя-версия-релиз}.{архитектура}.rpm`



Например, для процессоров `e2kv6` имя пакета будет выглядеть так: `admc-0.17.0-alt1.e2kv6.rpm`, а для архитектурно-независимого пакета имя может выглядеть так: `icon-theme-simple-sl-2.7-alt4.noarch.rpm`.

Имя установленного пакета в системе может отличаться от имени файла пакета.

⁴https://www.opennet.ru/docs/RUS/rpm_guide/13.html

⁵https://uneex.ru/static/RedHatRPMGuideBook/rpm_guide-linux.html#16_html

3.4 Рабочее пространство для сборки RPM-пакетов

Рабочее пространство сборки rpm-пакета это структура файлов и каталогов. Эту структуру можно создать двумя способами — вручную или через утилиту `rpmdev-setuptree`.

Для подготовки ручным способом структуры каталогов выполните команду:

```
$ mkdir -p ~/RPM/{BUILD,SRPMS,RPMS,SOURCES,SPECS}
```

Альтернативный способ подготовки рабочей среды — утилита `rpmdev-setuptree`. Утилита входит в состав пакета `rpmdevtools` (см. раздел 3.2). Для подготовки структуры каталогов через утилиту `rpmdev-setuptree` выполните команду:

```
$ rpmdev-setuptree
```

Утилита создаст базовую структуру каталогов и файл `~/rpmmacros`, если его не существовало.

Для системы ALT расположение структуры каталогов⁶ по умолчанию определяется в файле `~/rpmmacros` и находится в каталоге `~/RPM`:

```
$ tree ~/RPM
/home/user/RPM
├── BUILD
├── RPMS
├── SOURCES
├── SPECS
└── SRPMS
```

```
5 directories, 0 files
```

1. **BUILD** — в каталог попадают распакованные исходные файлы из **SOURCES** с уже применёнными патчами — стадия `%prep`. В каталоге **BUILD** происходит сборка программного обеспечения.
2. **RPMS** — в каталог **RPMS** попадают бинарные **rpm**-пакеты после сборки, в соответствии с подкаталогами для поддерживаемых архитектур.
3. **SOURCES** — в каталоге размещают архивы исходных данных и патчи.
4. **SPECS** — в каталоге размещают **spec**-файлы пакетов для сборки.
5. **SRPMS** — в каталог **SRPMS** попадают результаты сборки **srpm**-пакетов.

Созданная структура каталогов становится рабочей областью упаковки **rpm**-пакета.

⁶<https://rpm-packaging-guide-ru.github.io/#rpm-packaging-workspace>



В структуре сборочного окружения RPM существует понятие **buildroot**. Это каталог `/TMP`, в который попадают служебные файлы в ходе сборки пакета и уже подготовленные бинарные данные для упаковки в соответствии со структурой каталогов. По умолчанию **buildroot** создаётся во временном системном каталоге, но может быть переназначен.

3.5 Описание SPEC-файла

Спец-файл — RPM Specification File — это текстовый файл, который описывает процесс сборки и конфигурацию пакета, служит инструкцией для утилиты **rpmbuild**. Он содержит метаданные, такие как имя пакета, версию, лицензию, а также разделы с инструкциями для сборки, установки и упаковки программного обеспечения, журнал изменений пакета⁷.

Спец-файл можно рассматривать как «инструкцию», которую утилита **rpmbuild** использует для сборки **rpm**-пакета. **Спец-файл** состоит из трёх разделов: **Header** (Заголовок/Преамбула), **Body** (Тело) и **Changelog** (Журнал изменений).

1. **Header** (Заголовок) — этот раздел содержит метаданные о пакете, такие как его имя (Name), версия (Version), релиз (Release), краткое описание (Summary), лицензия (License) и другие параметры, которые идентифицируют и характеризуют пакет.
2. **Body** (Тело) — этот раздел содержит инструкции для процесса сборки пакета. В нём определяются различные секции, такие как **BuildRequires** (зависимости для сборки), **%build** (инструкции для сборки), **%install** (инструкции для установки), **%files** (список файлов, включённых в пакет) и другие.
3. **Changelog** (Журнал изменений) — этот раздел содержит историю изменений пакета. Он содержит записи о внесённых изменениях, включая дату изменения, автора и краткое описание того, что было изменено.

Сборочная зависимость пакета (зависимость для сборки) — вид зависимостей пакета, необходимых в процессе его сборки. Компоненты такого пакета не участвуют в работе пакета после его установки в систему и используются только на этапе сборки.

Заголовок (Преамбула)

Заголовок **спец-файла** содержит информацию о пакете: версию, исходный код, патчи, зависимости.

Рекомендуемый порядок заголовочных тегов:

⁷<https://www.altlinux.org/Spec>

- Name, Version, Release, Serial;
- далее Summary, License, Group, Url, Packager, BuildArch;
- потом Source[номер архива с исходными данными], Patch[номер патча];
- далее PreReqs, Requires, Provides, Conflicts;
- и, наконец, Prefix, BuildPreReqs, BuildRequires.

Ниже приведён пример части `spec`-файла `notepadqq`:

```
Summary:   A Linux clone of Notepad++
Name:      notepadqq
Version:   1.4.8
Release:   alt2
License:   GPLv3
Group:     Editors
URL:       http://notepadqq.altervista.org/wp/
Source0:   %name-%version.tar
Source1:   codemirror.tar
```

Тело

Тело `spec`-файла отвечает за выполнение сборки, установки или очистки пакета.

Описание структуры:

- В секции `%prep` производится распаковка архивов с исходными кодами и формируется директория с источниками⁸.
 - Макрос `%setup` перемещает файлы и распаковывает архивы с исходными данными в каталог для сборки, выполняет переход в распакованный каталог (источники описаны в преамбуле в тегах `Source`[номер архива с исходными данными]).
 - Макрос `%patch[номер патча]` инструкции для индивидуального применения патча с номером (патчи описаны в преамбуле в тегах `Patch`[номер патча]).
- В секции `%build` внутри ранее подготовленной директории производится сборка программы. Если это компилируемый язык, то исходные данные компилируются в бинарные файлы. Если это интерпретируемый язык, то процесс может не подразумевать компиляцию. Обычно за процесс сборки отвечают системы сборки, отличающиеся для разных языков программирования. Для C/C++ обычно используется `automake/autoconf` и макросы `%configure` и `%make_build`. Есть и другие системы сборки с другими макросами — `CMake`, `meson`, `pyproject` и т. д.

⁸https://www.opennet.ru/docs/RUS/rpm_guide/48.html

- В секции `%install` подготавливается новая директория с теми файлами, которые будут помещены в `rpm`-пакет в конце процесса сборки. Эта директория обозначается макросом `%buildroot`. Из текущей директории подготовленные на предыдущем этапе файлы (бинарные файлы, файлы документации, конфигурационные файлы и т.д.) нужно перенести в `%buildroot`. Например, файл `build/application.bin` нужно перенести в `%buildroot/usr/bin/application.bin`. За это в некоторых случаях может также отвечать система сборки, например, `automake/autoconf` так умеет, запускается через макрос `%makeinstall_std`. Для других систем сборки есть другие макросы.
- Возможно добавление секции `%clean`. Её задача очистить дерево сборки и каталог установки.
- Если разработчик добавляет в `спес`-файл собственные скрипты, их следует распределять в секции:
 - `%pre` (выполнение перед установкой);
 - `%post` (выполнение после установки);
 - `%preun` (перед удалением пакета);
 - `%postun` (после удаления пакета).
- Секция `%files` содержит список путей и файлов, которые будут упакованы в `rpm`-пакет и в дальнейшем установлены в систему.
 - В этой секции можно создать каталог (`%dir`), отметить, что файл является документацией (`%doc`) или файлом конфигурации (`%config`), или файл не относится к пакету, но необходим в начале работы приложения (`%ghost`).



Сборка `rpm`-пакета выполняет все инструкции, указанные в `спес`-файле. В процессе сборки утилита `rpmbuild` выводит на экран информацию о процессе сборки. С помощью этой информации можно отследить возможные ошибки описания `спес`-файла.

Журнал изменений

Секция журнал изменений (`%changelog`) содержит оформленный в принятом сообществом формате список с описанием изменений, проводимых с исходными данными и сборочными инструкциями пакета.



В пакете `rpm-utils` присутствует утилита `add_changelog`. Утилита добавит заголовок для журнала изменений на основе данных в преамбуле `спес`-файла.

Пример использования утилиты `add_changelog`:

```
$ add_changelog имя_спецификации.spec
```

3.6 Пример .спес-файла

Представим образцы `спес-файлов`⁹: шаблонный образец, образец для программы со сборочной системой `autotools`, образец для программы со сборочной системой `сmake` и образец модуля для `Python 3`.

Пример 1. Пустой СПЕС

```
Name:      <имя-пакета>
Version:   <версия-пакета>
Release:   alt<релиз-пакета>

Summary:   <однострочное описание>
License:   <лицензия>
Group:     <группа>

Url:       <URL>
Source:    %name-%version.tar
Patch1:

PreReq:
Requires:
Provides:
Conflicts:

BuildPreReq:
BuildRequires:
%{?!_without_test:%{?!_disable_test:%{?!_without_check:%{?!_disable_check:BuildRequires: }}}
BuildArch:

%description
<многострочное
описание>

%prep
%setup
%patch1 -p1

%build
%configure
%make_build

%install
%makeinstall_std

%check
%make_build check

%files
%_bindir/*
%_mandir/*
%doc AUTHORS NEWS README

%changelog
* <дата> <ваше имя> <$login@altlinux.org> <версия_пакета>-<релиз_пакета>
- initial build for ALT Linux Sisyphus
```

⁹<https://www.altlinux.org/SampleSpecs>

Пример 2. Для программы на autotools

Название GNU Autotools обычно относится к программным пакетам Autoconf, Automake, Libtool и Gnulib. Вместе они составляют систему сборки GNU. Этот спец-файл является примером пакета с программой.

```
Name:    sampleprog
Version: 1.0
Release: alt1

Summary: Sample program specfile
Summary(ru_RU.UTF-8): Пример спец-файла для программы

License: GPLv2+
Group: Development/Other
Url: http://www.altlinux.org/SampleSpecs/program

Source: %name-%version.tar
Patch0: %name-1.0-alt-makefile-fixes.patch

%description
This specfile is provided as sample specfile for packages with programs.
It contains most of usual tags and constructions used in such specfiles.

%description -l ru_RU.UTF-8

%prep
%setup
%patch0 -p1

%build
%configure
%make_build

%install
%makeinstall_std
%find_lang %name

%files -f %name.lang
%doc AUTHORS ChangeLog NEWS README THANKS TODO contrib/ manual/
%_bindir/*
%_mandir/*

%changelog
* Sat Sep 33 3001 Sample Packager <sample@altlinux.org> 1.0-alt1
- initial build
```

Пример 3. Для программы на cmake

CMake — это кроссплатформенный инструмент с открытым исходным кодом, который использует независимые от компилятора и платформы файлы конфигурации. **CMake** позволяет создавать собственные файлы инструментов сборки, специфичных для вашего компилятора и платформы. **CMake** считается стандартом де-факто для сборки кода на C++.

```
Name:    sampleprog
Version: 1.0
Release: alt1

Summary: Sample program specfile
License: GPLv2+
Group:   Development/Other
```



```

Url: http://www.altlinux.org/SampleSpecs/cmakeprogram
Source: %name-%version.tar.bz2

BuildRequires(pre): cmake rpm-macros-cmake

%description
This specfile is provided as a sample specfile
for a package built with cmake.

%prep
%setup

%build
%cmake_insource
%make_build # VERBOSE=1

%install
%makeinstall_std
%find_lang %name

%files -f %name.lang
%doc AUTHORS ChangeLog NEWS README THANKS TODO contrib/ manual/
%_bindir/*
%_mandir/*

%changelog
* Sat Jan 33 3001 Example Packager <example@altlinux.org> 1.0-alt1
- initial build

```

Пример 4. Модуль для Python 3

Макросы для сборки модулей python3 содержатся в пакете rpm-build-python3 и аналогичны тем, что используются в ALT для python¹⁰.

```

%define pypi_name @NAME@
%define mod_name %pypi_name

%def_with check

Name: python3-module-%pypi_name
Version: 0.0.0
Release: alt1
Summary: @TEMPLATE@
License: MIT
Group: Development/Python3
Url: https://pypi.org/project/@NAME@
Vcs: @SOURCE_GIT@
BuildArch: noarch
Source: %name-%version.tar
Source1: %pyproject_deps_config_name
Patch: %name-%version-alt.patch

# mapping of PyPI name to distro name
Provides: python3-module-{{pep503_name %pypi_name}} = %EVR

%pyproject_runtimedeps_metadata
BuildRequires(pre): rpm-build-pyproject
%pyproject_builddeps_build
%if_with check
%pyproject_builddeps_metadata
%endif

```

¹⁰<https://www.altlinux.org/Python3>


```

%description
@DESCR@

%prep
%setup
%autopatch -p1
%pyproject_deps_resync_build
%pyproject_deps_resync_metadata

%build
%pyproject_build

%install
%pyproject_install

%check
%pyproject_run_pytest

%files
%doc README.*
%python3_sitelibdir/%mod_name/
%python3_sitelibdir/{%pyproject_distinfo %pypi_name}

%changelog

```

3.7 RPM макросы

Макрос RPM — это именованная переменная, которая напрямую подставляет текст в `спес`-файл во время сборки `rpm`-пакета. Имена макросов начинаются с символа `%`. Имена макросов — это сокращённые псевдонимы для часто используемых фрагментов текста.

Ниже приведены примеры макросов¹¹:

1. **Пример макроса, содержащего значение.** Если во время сборки некоторым командам необходимо передать имя собираемого пакета, то можно передавать им макрос `%name`. Во время сборки этот макрос подставляет имя пакета, объявленное в начале `спес`-файла:

```

%define some_macro foo
....
Name: bar-%some_macro
.....
%build
%dune_build -p %some_macro

```

2. **Пример макроса с набором команд.** `%cmake_build` — макрос, необходимый для сборки пакетов с помощью `cmake`. Он подставляет следующую последовательность команд:

```

mkdir -p e2kv6-alt-linux-gnu;
cmake \

```

¹¹https://www.altlinux.org/Спец/Предопределённые_макросы


```
-DCMAKE_SKIP_INSTALL_RPATH:BOOL=yes \  
-DCMAKE_C_FLAGS:STRING='-O2 -g' \  
-DCMAKE_CXX_FLAGS:STRING='-O2 -g' \  
-DCMAKE_Fortran_FLAGS:STRING='-O2 -g' \  
-DCMAKE_INSTALL_PREFIX=/usr \  
-DINCLUDE_INSTALL_DIR:PATH=/usr/include \  
-DLIB_INSTALL_DIR:PATH=/usr/lib64 \  
-DSYSCONF_INSTALL_DIR:PATH=/etc \  
-DSHARE_INSTALL_PREFIX:PATH=/usr/share \  
-DLIB_DESTINATION=lib64 \  
-DLIB_SUFFIX="64" \  
-S . -B "e2kv6-alt-linux-gnu"
```

Преимущества использования макросов:

- упрощение сборки;
- унификация `spec`-файлов;
- подбор шаблонов для создания `spec`-файлов;
- сокращение размера `spec`-файлов позволяет упростить отладку;
- использование макросов обеспечивает гибкость в настройке и конфигурации пакетов, позволяя быстро изменять параметры сборки.

Где объявлены `rpm`-макросы:

- стандартные макросы предопределены в пакете `rpmbuild (librpm)`. Информацию о них можно получить из файла `/usr/lib/rpm/macros` или выполнив команду: `rpm --showrc`;
- макросы можно объявить самостоятельно, добавив в `spec`-файл;
- макросы можно объявить в отдельных файлах. Команда `%include` позволяет загрузить специальные файлы с объявленными макросами;
- макросы, объявленные в файлах, поставляемые с другими пакетами.

Файлы с объявленными `rpm`-макросами хранятся в каталоге `/usr/lib/rpm/macros.d`.

Например, пакет `rpm-build-ruby` содержит готовые макросы для сборки пакетов с программами, написанными на языке `Ruby`. Для того, чтобы использовать эти макросы, необходимо этот пакет добавить в зависимости: `BuildRequires(pre): rpm-build-ruby`.



Команда получения значения макроса: `rpm --eval <имя_макроса>`



Некоторые макросы могут быть вложенными.

3.8 Вопросы для самопроверки

1. Какие инструменты (программы) нужны для сборки пакетов в «Альт Платформа»?
2. Какая утилита используется для непосредственной сборки бинарного пакета?
3. Верно ли что **hasher** обеспечивает воспроизводимую сборку в изолированной среде?
4. Каково внутреннее устройство **rpm**-пакета?
5. У вас есть пакет `admc-0.15.0-alt1.x86_64.rpm`:
 - а) можно ли его установить на компьютер с процессором Байкал-М?
 - б) а на компьютер с процессором Эльбрус 16С?
 - в) какой пакет нужно взять, чтобы собрать тот же пакет под другую целевую архитектуру?
6. Какая структура каталогов необходима для сборки **rpm**-пакета?
7. Понятие **buildroot** в пространстве сборки **rpm**-пакетов, какое назначение?
8. Для чего служит **spec**-файл?
9. Из каких обязательных частей состоит **spec**-файл?
10. Какие зависимости можно называть «для сборки», а какие «времени исполнения»?
 - а) В какой части **spec**-файла указывается имя пакета?
 - б) В какой части **spec**-файла указывается лицензия под которой распространяется пакет?
 - в) В какой части **spec**-файла указывается история изменений пакета?
11. Что такое **rpm**-макрос?

- a) Какие преимущества даёт использование макросов?
- b) Как найти стандартные макросы?
- c) Где в системе находятся макросы, поставляемые сторонними пакетами?

Глава 4

Инструмент `rpmbuild`

rpmbuild — утилита сборки **rpm**-пакета из набора подготовленных файлов. Сборка осуществляется в локальном окружении операционной системы, то есть в процессе сборки используются пакеты, установленные в системе. **rpmbuild** позволяет собрать пакет с исходными данными (или в частном случае пакеты с исходным кодом программ) и пакеты с бинарным содержимым.

Для работы с утилитой необходимо установить пакет **rpm-build**:

```
$ apt-get install rpm-build
```

Общий вид структуры вызова утилиты:

```
$ rpmbuild [параметры]
```

Для подробной информации вызовите справку командой:

```
$ rpmbuild --help
```

4.1 Описание инструмента и сборка **rpmbuild**

Параметры сборки **rpmbuild**:

- **-ba** — сборка исходного и двоичного пакета по файлу спецификации;
- **-bb** — сборка только бинарного пакета по файлу спецификации;
- **-bs** — сборка только пакета с исходными данными (**src.rpm**) по файлу спецификации;
- **-tb** — сборка бинарного пакета из **tar**-архива;
- **-ts** — сборка исходного пакета из **tar**-архива;

- `-b1` — проверить список файлов в каталогах дерева окружения и вывести ошибки, если отсутствуют необходимые файлы;
- `--rmsource` — удаление исходных файлов после завершения сборки;
- `--rmspec` — удаление файла спецификации после завершения сборки.

Утилита предполагает наличие подготовленной структуры каталогов для сборки — дерево каталогов. (см. 3.4 «Рабочее пространство для сборки RPM-пакетов»). По умолчанию используется каталог `~/RPM`.

```
$ tree ~/RPM
/home/user/RPM
├── BUILD
├── RPMS
├── SOURCES
├── SPECS
└── SRPMS
```

5 directories, 0 files

Выбранные для сборки исходные данные необходимо упаковать в `.tar`-архив:

```
$ tar -cvf <ИМЯ ФАЙЛА АРХИВА>.tar <КАТАЛОГ ИСХОДНЫХ ДАННЫХ>/
```

Архив с исходными данными для сборки нужно поместить в каталог:
`./RPM/SOURCES/`.

Подготовленный `.spec` файл с инструкциями поместить в каталог:
`./RPM/SPECS`.

```
$ cp <ИМЯ ФАЙЛА АРХИВА>.tar ~/RPM/SOURCES/<ИМЯ ФАЙЛА АРХИВА>.tar && \
cp <ИМЯ СПЕС-ФАЙЛА>.spec ./RPM/SPECS/<ИМЯ СПЕС-ФАЙЛА>.spec
```

Пример сборки бинарного пакета `.rpm`:

```
$ rpmbuild -bb ~/RPM/SPECS/<ИМЯ СПЕС-ФАЙЛА>.spec
```

После успешного выполнения процесса сборки бинарные пакеты `.rpm` помещаются в каталог `~/RPM/RPMS/`.

Пример сборки пакета с исходными данными `.src.rpm`:

```
$ rpmbuild -bs ~/RPM/SPECS/<ИМЯ СПЕС-ФАЙЛА>.spec
```

После успешного выполнения процесса сборки, пакеты `.src.rpm` помещаются в каталог `~/RPM/SRPMS/`.

Рассмотрены два базовых сценария сборки пакета. Для получения различных результатов можно комбинировать ключи. Например, для сборки и бинарного пакета и пакета с исходными данными можно использовать ключ `-ba`. Полученные пакеты можно переносить и устанавливать на различные компьютеры любым доступным способом. В общем виде сборка пакета состоит из этапов:

- установка пакета `rpm-build`;
- подготовка окружения сборки;
- формирование исходных данных;
- упаковка исходных данных для сборки в `.tar`-архив;
- подготовка файла спецификации для сборки `.spec`;
- размещение подготовленных файлов в каталогах окружения сборки;
- запуск локальной сборки.

Сборка пакетов в локальном окружении операционной системы вызывает некоторые неудобства. Приходится вручную раскладывать файлы в структуре каталогов `~/RPM`. При изменении исходных данных приходится заново создавать архив. Работа в локальном окружении потенциально может вызвать проблемы безопасности в случае сборки и исполнения заранее не проверенных исходных данных собираемых программ и компонентов. Все дополнительные компоненты, необходимые в процессе сборки, следует устанавливать в операционную систему так же в ручную. В случае разработки дистрибутива операционной системы такие недостатки неприемлемы. Поэтому разработчики дистрибутивов операционных систем создают свои собственные инструменты, позволяющие исключить недостатки локальной сборки пакетов.

4.2 Вопросы для самопроверки

1. Что такое `rpmbuild`, для чего предназначен?
2. Какая структура каталогов необходима для начала сборки пакета средствами `rpmbuild`?
3. Какие основные файлы необходимы для начала сборки пакета средствами `rpmbuild`?
4. Какие два вида пакетов можно получить средствами `rpmbuild`?
5. Перечислите основные этапы сборки пакета.

Глава 5

Инструмент Hasher

Hasher — инструмент для сборки пакетов с использованием изолированной среды (**chroot**). Представляет собой сложный усовершенствованный инструмент, направленный на устранение недостатков, связанных с процессом сборки пакетов в локальной среде средствами **rpm-build**.

- Благодаря входящим в комплект утилитам упрощается процесс поддержания сборочных зависимостей и сохранение целостности пакета.
- Изолированная среда обновляется для каждой сборки, инициированной средствами **Hasher**, что гарантирует независимый от конфигурации операционной системы процесс сборки пакета.
- Изменяя конфигурацию источников для **Hasher**, указывая различные репозитории, возможно собирать пакеты для пакетных баз отличных от локальной операционной системы.

Для начала работы необходимо установить пакет **hasher** и настроить работу с утилитой:

```
# apt-get install hasher
```

Возможности **Hasher** задокументированы¹.

```
$ man hsh
```

```
$ man hasher-priv
```

5.1 Настройка Hasher

Для начала работы необходимо настроить утилиту:

- Проверьте версию пакета **hasher-priv**:

¹http://uneex.ru/static/AltlinuxOrg_Hasher/


```
$ rpm -qa hasher-priv
```

Если версия **hasher-priv** старше 2.0, то запустите демона:

```
# systemctl enable --now hasher-privd.service
```

- Подготовить учётную запись для работы с **hasher** можно средствами встроенной утилиты. Пользователь должен отличаться от корневого супер пользователя (**root**) и должен быть лишен повышенных привилегий для обеспечения безопасности системы от процесса сборки. Утилита создаст дополнительных пользователей добавит в необходимые группы и настроит **hasher** для работы от указанного пользователя:

```
# hasher-useradd <имя учетной записи>
```



Если утилитой **hasher-useradd** настраивается активный пользователь в системе, то поскольку утилита производит манипуляции с добавлением пользователя группы, пользователю необходимо создать новую или перезапустить рабочую сессию, чтобы изменения вступили в силу.

- Для перезапуска рабочей сессии пользователя **user** применяется команда:

```
# su - user
```

- Создать каталоги расположения сборочной среды и каталоги настроек (следующие указанные пути используются утилитой по умолчанию):

```
$ mkdir ~/hasher
$ mkdir ~/.hasher
```

- Добавить ваше имя (только латинские символы) в указанном формате в конфигурационный файл:
\$ echo 'packager="Имя Фамилия <my_mail@altlinux.org>"' >> ~/.hasher/config

Этот параметр важен для встроенных проверок **hasher**.

- Добавить отключение излишних проверок в условиях локальной сборки:
\$ echo 'no_sisyphus_check="packager,buildhost,gpg,changelog"' >> ~/.hasher/config

При создании каталога **hasher** следует учитывать два правила:

1. права доступа соответствуют **drwxr-xr-x**, то есть каталог доступен на запись;
2. на файловой системе, смонтированной с **noexec** или **nodev**, каталог располагать нельзя²:
 - **noexec** устанавливается, если в системе есть файлы с правами **rwsr-xr-x** (запустить исполняемые файлы с правами владельца или группы) и владельцем **root**. Запустить файл **rw-r-xr-x** на такой файловой системе невозможно, а следовательно, и создать корректное сборочное окружение. **Hasher** не зависит от пользовательского окружения.
 - **nodev** говорит о том, что на файловой системе не будут созданы файлы устройств. Это не соответствует функциональности **hasher** (см. ниже в данной главе про **fstab**).



Проверьте права доступа файлов источников для пакетов системы **/etc/apt/**. Хотя бы один источник должен быть доступен на чтение для группы **hashman**. Для назначения группы на стандартные файлы источников можно выполнить команду:

```
# chgrp hashman /etc/apt/sources.list{, .d/*}
```

Если всё сделано верно, мы сможем создать минимальное сборочное окружение, исполнив команду:

```
$ hsh --initroot-only --verbose
```

По умолчанию сборочное окружение создается в каталоге **~/hasher**. Для инициализации окружения этот каталог должен существовать. Можно переобозначить путь для сборочного окружения по умолчанию. Для этого нужно создать каталог, в котором планируем расположить новое сборочное окружение и выполнить команду с переобозначенным путем:

```
$ mkdir ~/some-new-hashier
$ hsh --initroot-only ~/some-new-hashier
```

Окружение сбрасывается для каждого нового запущенного процесса сборки, и создается заново, если не существовало. При необходимости содержимое каталога может быть очищено с правами супер пользователя.

Часто используемые параметры конфигурации **~/hasher/config**:

- **no_sisyphus_check="packager,buildhost,gpg"** — рекомендуемый параметр для работы в локальном окружении. Отключает проверки излишних значений в условиях локальной сборки;

²<https://serverfault.com/questions/547237/explanation-of-nodev-and-nosuid-in-fstab>

- `packager=" rpm --eval %packager "` — альтернативный вариант указать параметр `packager` для `rpm`. (Будет работать если заполнен макрос `%packager` в `~/.rpmmacros`);
- `allowed_mountpoints=/dev/pts,/proc,/dev/shm` — параметр, позволяющий подключать в корень изолированного сборочного окружения список локальных файловых систем;
- `lazy_cleanup=1` — параметр очищает среду сборки перед каждой новой сборкой;
- `apt_config="$HOME/.hasher/apt.conf"` — параметр позволяет переобозначить конфигурацию источников репозитория для сборочного окружения.

5.2 Описание системы Hasher

Переместиться в корень сборочного окружения можно выполнив команду:

```
$ hsh-shell
```

Приведенная команда перемещает командную оболочку в изолированную среду `hasher`. Команда позволяет перемещаться по структуре изолированного окружения и продолжить работу внутри.

Опишем структуру каталогов `hasher`.

- `~/hasher`
 - `chroot` — сборочное окружение. В этом каталоге находится корневое дерево содержащее минимальный набор пакетов, необходимых для сборки.
 - `aptbox` — набор утилит для установки, обновления и удаления пакетов `chroot`. Например, тут лежит модифицированный `apt-get`, с помощью которого происходит установка пакетов в `chroot`.
 - `cache` — в этом каталоге хранятся временные файлы, необходимые для создания `chroot`.
- `repo`, который содержит подкаталоги:
 - `SRPMS.hasher` — пакеты с исходными данными (`sources`).
 - `<архитектура>/RPMS.hasher/` — каталог с пакетами, собранными под конкретную архитектуру. Содержимое каталога дополняет источники пакетов для сборочного окружения. Помещенные в него пакеты можно установить в изолированном окружении `hasher`. Такой механизм позволяет добавлять пакеты в сборочные зависимости и использовать пакеты в изолированной среде, которые ещё не существуют в подключенных репозиториях.

- `~/hasher` — каталог конфигурационных файлов. Каталог может отсутствовать, в этом случае `hasher` использует конфигурацию по умолчанию.
 - `apt.config` — конфигурация для `apt-get` из `~/hasher/aptbox/`.
 - `config` — конфигурация самого `hasher`.
- `/etc/hash-priv/` каталог с конфигурацией для вспомогательной утилиты `hash-priv`.
 - `./user.d` — каталог содержит файлы настроенных пользователей для работы с `hasher`.
 - `fstab` — информация о точках монтирования вспомогательной программы `hash-priv system` — конфигурация вспомогательной программы `hash-priv`.

В структуре каталогов `hasher` стоит обратить внимание на служебные подкаталоги, позволяющие взаимодействовать с локальной системой:

- `~/hasher/chroot/.in` — предполагает добавление файлов из локальной системы.
- `~/hasher/chroot/.out` — предполагает получение файлов из `chroot` системы.

Непосредственно структура каталогов `rpmbuild` сборки находится:

```
~/hasher/chroot/usr/src/RPM
```

`Hasher` умеет монтировать внутрь изолированной среды виртуальные файловые системы из локальной машины³. Этот механизм применяется в тех случаях, когда собираемому приложению для сборки требуется доступ к ресурсам основной машины, которые `Hasher` не предоставляет по умолчанию. Например, виртуальная файловая система `/proc` или `/dev/pts`, которых по умолчанию нет в `hasher`-контейнере. Файловая система `/proc` получает информацию о состоянии и конфигурации ядра и системы.

Для монтирования файловой системы следует:

1. В файле `/etc/hash-priv/fstab` описать файловую систему.
2. В файле `/etc/hash-priv/system` указать файловую систему с помощью опции `allowed_mountpoints`.
3. Указать файловую систему либо при запуске `Hasher` в опции `--mountpoints`, либо в конфигурационном файле `~/hasher/config` в ключе `known_mountpoints`.
4. Прописать необходимую файловую систему в `спес`-файле в теге `BuildReq`, либо в списке зависимостей.

³<https://www.altlinux.org/Hasher/Руководство>

5.3 Сборка в Hasher

Сборка выполняется от предварительно настроенного пользователя (см. раздел 5.1 «Базовая настройка Hasher»).

Изначально сборка в **hasher** предполагает наличие подготовленного **.src.rpm**-пакета (см. раздел 3.3 «Описание RPM-пакета»), или специально подготовленного **tar**-архива.

Для запуска сборки необходим подготовленный файл **.src.rpm** (см. раздел 4 «Пример сборки пакета с исходными данными **.src.rpm**»), после чего можно запустить сборку, выполнив команду:

```
$ hsh ./<имя пакета>.src.rpm
```

Для сборки можно передать специально подготовленный архив **.tar**. Приведем упрощенный пример использования архива для сборки.

Пример структуры каталога для архива:

```
./
├── <имя пакета>.spec
├── <имя патча>.patch
├── <имя пакета>-<версия из spec>.tar
└── <имя пакета>-<версия из spec> - Каталог с исходными данными для пакета.
```

Упрощенный пример команды для создания **tar**-архива:

```
$ tar --create --file=pkg.tar --label=<имя пакета>.spec \
<имя пакета>-<версия из spec>.tar <имя патча>.patch <имя пакета>.spec
```

Подготовленный архив можно запустить на сборку, выполнив команду:

```
$ hsh ./pkg.tar
```

Собранный бинарный пакет появляется в директории:

```
~/hasher/repo/<архитектура>/RPMS.hasher/.
```

Собрать пакет в изолированной среде можно вручную из исходных данных, инструмент **rpmbuild** (пример работы в главе 4 «Инструмент **rpmbuild**»). Для этого подготовленный файл **.spec** и архив с исходными данными нужно поместить в сборочное окружение, воспользовавшись служебным каталогом **~/hasher/chroot/.in**, и провести сборку **rpmbuild** внутри подготовленного сборочного окружения.

Из-за своей сложности инструмент не используется отдельно и приведен для ознакомления и подготовки к работе в дальнейшем. Полноценно функционал инструмента раскрывается в связке с **GEAR** (инструмент **GEAR** будет описан в следующей главе).

5.4 Вопросы для самопроверки

1. Что такое **Hasher** и для чего он предназначен?
2. Какова структура каталогов **hasher**?
3. Почему ранее собранные в **hasher** пакеты не оказывают влияния на новую сборку?
4. Можно ли указать из какого репозитория будет происходить сборка в **hasher**?
5. Какие шаги, помимо установки, необходимо сделать для настройки **hasher**?
6. Какие сценарии сборки возможны при использовании **hasher**?
7. Как посмотреть справочную информацию по пакету **hasher** и **hasher-priv**?
8. Зачем монтировать внутрь **hasher** файловые системы?

Глава 6

Инструмент GEAR

GEAR (Get Every Archive from git package Repository) — инструмент для подготовки, поддержки и сборки пакетов в **git**-репозитории^{1,2}.



Для работы с текущим разделом необходимо изучить документацию по работе с распределённой системой управления версиями **Git**.

Git — система контроля версий, предназначенная для хранения файлов, истории их изменений и служебной информации.

git-репозиторий — набор файлов, находящийся под управлением системы контроля версий **Git**.

gear-репозиторий — **git**-репозиторий, дополненный **gear** инструкциями для подготовки исходных данных для последующей сборки пакета.

- Расширенный набор утилит упрощает процессы сборки пакетов на всех этапах работы с исходными данными.
- **GEAR** интегрирован с инструментами **rpmbuild** и **hasher**.
- Работа в **git**-репозитории добавляет плюсы системы контроля версий и предоставляет возможность воспроизводимой сборки пакета.
- **GEAR** даёт возможности гибкой настройки репозитория исходных данных.

Для начала работы необходимо установить пакет **gear**:

```
# apt-get install gear
```

Подробная инструкция базовых компонентов:

¹<https://www.altlinux.org/Gear>

²<https://www.altlinux.org/Gear/Справочник>


```
$ man gear
```

```
$ man gear-rules
```

6.1 Описание GEAR

При сборке пакетов **gear** предлагает работать в том же **git**-репозитории, в котором хранятся исходные данные пакета. **GEAR** позволяет целиком импортировать историю разработки, предоставляет различные средства импорта исходных данных и различные варианты организации репозитория.

Идея **GEAR** в доступности всего необходимого для сборки пакета в **git**-репозитории либо в репозитории пакетов, на основе которого ведётся сборка. Система контроля версий **Git** предоставляет встроенные механизмы обеспечения целостности (контрольные суммы, криптографически подписанные теги) для задач управления пакетами. Появляется возможность воспроизводимой сборки³ — собрать «такой же» пакет ещё раз, опираясь на логически законченную версию изменений, зафиксированную на момент времени⁴.

Для удобства сборки пакетов **gear** интегрирован с инструментами **rpm-build** и **hasher**. Из **gear**-репозитория можно одной командой собрать пакет при помощи **rpmbuild** или **hsh**.

Сборка пакета ведётся из конкретного коммита, который мы в дальнейшем будем называть главным. Именно в нём должна находиться нужная версия **.spec**-файла и правил экспорта. **GEAR** позволяет экспортировать из **git**-репозитория каталоги и подкаталоги в виде архивов, экспортировать отдельные файлы, вычислять разницу (**diff**) и сохранять её в виде патча. При этом может использоваться как состояние в главном коммите, так и в любом из его предков, прямых и не прямых. Это позволяет гибко и удобно организовать работу по поддержке пакета. Все нужные исходные данные можно хранить в **git** так, чтобы с ними было удобно работать, а затем экспортировать так, чтобы ими было удобно воспользоваться из **.spec**-файла.

6.2 Правила экспорта

Правила экспорта для **GEAR** описываются в текстовом файле, который также хранится в репозитории. По умолчанию **GEAR** ищет этот файл по пути **.gear/rules** или **.gear-rules**, от корня репозитория, в главном коммите.

Этот файл состоит из одной или нескольких строк, каждая из которых имеет следующий формат: **<директива>: <параметры>**.

Параметры разделяются пробельными символами. Пустые строки и строки, начинающиеся с «**#**», игнорируются.

В значениях многих параметров и опций директив могут применяться ключевые слова:

³https://www.altlinux.org/Воспроизводимая_сборка

⁴коммит (commit) — законченная версия изменений в терминологии **git**.

- `@name@` — будет заменено на имя пакета (извлекается из `.spec`-файла);
- `@version@` — будет заменено на версию пакета (извлекается из `.spec`-файла);
- `@release@` — будет заменено на релиз пакета (извлекается из `.spec`-файла).

Теги и пути

По умолчанию все пути в аргументах директив считаются от корня репозитория главного коммита. Однако большинство директив позволяет указать другой коммит в качестве основы. Для этого путь должен быть передан в формате:

`base_tree:path_to_file.`

В качестве `base_tree` может выступать:

- полный идентификатор коммита (SHA-1, 40 шестнадцатиричных цифр);
- имя тега `GEAR`;
- символ «.», обозначающий главный коммит.

В любом случае коммит, на который так ссылаются, должен быть предком главного коммита.

Основные директивы

Ниже приведены основные директивы `GEAR` и их аргументы. Подробнее с ними можно ознакомиться в `man gear-rules`.

- `spec: <путь>`

Задаёт путь к `.spec`-файлу. По умолчанию `GEAR` использует файл с расширением `.spec` из корня репозитория в главном коммите, если такой файл там только один. Единственный аргумент — путь к `.spec`-файлу.

- `copy: <glob> ...`

Скопировать файл, соответствующий указанному шаблону поиска (`glob pattern`). Может принимать несколько аргументов, для каждого из которых должны быть найдены соответствующие файлы.

Также существуют директивы `gzip`, `bzip2`, `lzma`, `lzma`, `zstd`, аналогичные `copy`, но сжимающие экспортированный файл подходящим алгоритмом сжатия.

- `tar: <tree_path>`

Экспортировать каталог из репозитория в виде `tar`-архива. Допустимые опции:

— `name=<archive_name>` имя архива (без суффикса `.tar`);

- **base=<base_name>** внутри архива будет создан каталог с указанным именем и все файлы будут помещены в него;
- **suffix=<suffix>** расширение создаваемого архива (по умолчанию — **.tar**);
- **exclude=<glob_patter>** не включать в архив файлы, соответствующие указанному шаблону поиска.

Помимо стандартных ключевых слов, в опциях **name** и **base** может применяться ключевое слово **@dir@**, которое будет заменено на имя каталога из параметра **tree_path**.

- **zip: <tree_path>**

Экспортировать каталог из репозитория в виде **zip**-архива. Принимает те же аргументы, что и директива **tar**, использует **.zip** в качестве расширения по умолчанию.

Также существуют директивы **tar.gz**, **tar.bz2**, **tar.lzma**, **tar.xz**, **tar.zst**, аналогичные **tar**, но сжимающие созданный архив подходящим алгоритмом сжатия. Чаще всего используются несжатые архивы, так как сжатия, используемого при сборке **SRPM**, обычно достаточно.

- **diff: <old_tree_path> <new_tree_path>**

Создать **unified diff** между указанными каталогами и сохранить его в виде патча. Допустимые опции:

- **name=<diff_name>** имя создаваемого файла;
- **exclude=<glob_patter>** игнорировать файлы, соответствующие указанному шаблону поиска.

Помимо стандартных ключевых слов в опции **name** могут применяться ключевые слова **@old_dir@** и **@new_dir@**, которые будут заменены на имя каталога из параметра **old_tree_path** и **new_tree_path** соответственно.

Все приведённые директивы требуют, чтобы все указанные в них файлы и каталоги существовали. Если какого-то файла или каталога не будет существовать, экспорт завершится ошибкой. Однако существуют аналогичные им директивы, заканчивающиеся знаком вопроса (например, **tar?:** или **copy?:**), которые игнорируют отсутствующие файлы. Это может быть удобно, чтобы не приходилось менять правила экспорта при добавлении и удалении патчей.

6.3 Основные типы устройства gear-репозитория

Гибкость **GEAR**⁵ означает, что каждый пользователь может настроить его по своему усмотрению. Существует несколько распространённых способов организации **gear**-репозитория в качестве основы для более сложных конфигураций.

⁵https://www.altlinux.org/Руководство_по_gear

Знакомство с ними поможет понять организацию репозитория при совместной работе над пакетами.

Базовые виды ведения GEAR репозитория:

- **Архив с исходными данными и патчи.**

Исходные данные хранятся в каталоге `package_name`; дополнительные изменения хранятся в виде патчей. Подобные репозитории создаёт команда `gear-srpmimport`, и также выглядят импортированные пакеты в `git.altlinux.org/srpms`.

В каталоге `package_name` принято хранить не изменённые исходные данные, для их обновления удобно использовать команду `gear-update`.

Такой формат будет больше всего знаком пользователям без опыта поддержки пакетов `source RPM` и при работе с проектами, не имеющими публичного `git`-репозитория или не использующими `git`.

Пример `.gear/rules`:

```
tar: package_name
copy?: *.patch
```

- **Репозиторий с историей исходного репозитория и модифицированными исходными данными.**

Создаётся копия исходного `git`-репозитория. Находится коммит, из которого нужно взять исходные данные для сборки. На основе этого коммита добавляется каталог `.gear` и `.spec`-файл. При обновлении новые исходные данные сливаются (`merge`) в эту ветку. Создаётся `gear`-тег, соответствующий собираемой версии. Изменения могут храниться в виде патчей, но чаще вносятся в текущую ветку со `spec`-файлом, затем изменения экспортируются в виде одного большого патча.

Пример `.gear/rules` с генерацией патча:

```
tar: v@version@:.
diff: v@version@:. . exclude=.gear/** exclude=*.sp
```

- **Пустая ветка со `.spec`-файлом и отдельная история разработки.**

Изначально `.spec`-файл и `.gear` ведутся в отдельной ветке, содержащей главный коммит. Исходные данные, необходимые для сборки пакета, сливаются (`merge`) при необходимости с `git` стратегией `ours` — таким образом, нужные коммиты оказываются в истории главного, но сами исходные данные в рабочее дерево каталога не попадают. Если нужно внести какие-то специфичные изменения, они вносятся в отдельную ветку, например `alt-fixes`. При каждом обновлении кода ветка `alt-fixes` и соответствующий ей `gear`-тег должны обновляться. В нашем примере `gear`-тег совпадает с именем ветки.

Пример `.gear/rules`:

```
tar: v@version@:.  
diff: v@version@:.. alt-fixes:..
```

6.4 Работа с GEAR

Импорт `.src.rpm`

Пакет в формате source RPM можно импортировать в `gear`-репозиторий командой `gear-srpmimport`:

```
$ mkdir package_name  
$ cd package_name  
$ git init -b sisyphus  
$ gear-srpmimport /путь/к/package_name.src.rpm
```

Получение готового репозитория с внешнего `git`-сервера

Достаточно клонировать репозиторий:

```
$ git clone <repository url> package_name  
$ cd package_name
```

Никаких дополнительных настроек не требуется.

Сборка пакета

Основным инструментом экспорта и сборки пакетов является команда `gear`. На практике удобно использовать предоставляемые команды-обёртки, а к самой команде `gear` прибегать только в самых сложных случаях. Командной обёртке можно передавать опции как утилиты `gear`, так и вложенной утилиты.

Командная обёртка `gear` для `rpmbuild` — `gear-rpm`. Для сборки пакета используйте команду:

```
$ gear-rpm --verbose -ba
```

Собрать пакет при помощи `hsh`:

```
$ gear-hsh --verbose
```

Команде `gear-hsh` можно передать как аргументы `gear`, так и аргументы `hsh`.

Если не указана опция `--tree-ish`, `gear`, в качестве главного коммита использует текущий коммит (`HEAD`). Если хочется проверить свежие изменения без создания коммита, можно использовать опцию `--commit`. Опция `--commit` доступна и для `gear`, и для `gear-rpm`, и для `gear-hsh`. В таком случае будет создан временный коммит (аналогично `git commit -a`), и пакет будет собран уже

из него. Стоит отметить, что, аналогично `git commit -a`, в таком коммите не будет новых файлов, если они ещё не были добавлены в `git` командой `git add`.

Сборка пакета из репозитория разработчиков

Для примера возьмём конкретный пакет, который уже есть в репозитории Сизиф, например `pixz`, и попробуем собрать его с нуля.

Для начала клонируем репозиторий. Сразу зададим имя удалённого репозитория:

```
$ git clone https://github.com/vasi/pixz -o upstream
$ cd pixz
```

Перейдём в ветку, из которой будем собирать пакет:

```
$ git checkout -b sisypheus upstream/master
```

Переместимся на тег (тег — это ссылка, указывающая на определённый `git`-коммит в репозитории), соответствующий версии, которую мы будем собирать. На момент написания пособия последний тег `v1.0.7`:

```
$ git reset --hard v1.0.7
```

Определим правила экспорта:

```
$ mkdir .gear
$ vim .gear/rules
```

Правила зададим следующие:

- `.spec`-файл перенесём в каталог `.gear`, чтобы он не путался с основными исходными данными;
- исходные данные будем забирать из `gear`-тега, соответствующего апстримному;
- сразу создадим патч, включающий все наши изменения (каталог `.gear` в этот патч мы включать не будем).

Получаем следующий файл `.gear/rules`:

```
spec: .gear/pixz.spec
tar: v@version@:..
diff: v@version@:.. . exclude=.gear/**
```

Создадим соответствующий версии `gear`-тег:

```
$ gear-store-tags v1.0.7
```

Напишем `.spec`-файл:


```
$ vim .gear/pixz.spec
```

```
%define _unpackaged_files_terminate_build 1
```

```
Name:      pixz
Version:   1.0.7
Release:   alt1
```

```
Summary:   Parallel, indexed xz compressor
License:   BSD-2-Clause
Group:     Other
Url:       https://github.com/vasi/pixz
```

```
Source:    %name-%version.tar
Patch:     %name-%version-%release.patch
```

```
BuildRequires: pkgconfig(liblzma) pkgconfig(libarchive)
BuildRequires: /usr/bin/a2x
```

```
%description
```

Pixz (pronounced *pixie*) is a parallel, indexing version of 'xz'.

The existing XZ Utils provide great compression in the '.xz' file format, but they produce just one big block of compressed data. Pixz instead produces a collection of smaller blocks which makes random access to the original data possible. This is especially useful for large tarballs.

```
%prep
%setup
%patch -p1
```

```
%build
%autoreconf
%configure
%make_build
```

```
%install
%makeinstall_std
```

```
%check
%make_build check
```

```
%files
%_bindir/*
%_mandir/*
%doc *.md
```



```
%changelog
* Wed Mar 31 2021 Author Name <email@altlinux.org> 1.0.7-alt1
- Initial build for Sisypheus
```

Важно, что версия в `спес-файле` — 1.0.7, поэтому конструкция `v@version@`, которую мы использовали в `.gear/rules`, раскроется в строку `v1.0.7` — именно такой тег `GEAR` мы создали.

Добавим каталог `.gear` в `git`:

```
$ git add .gear
```

Можно попробовать собрать пакет:

```
$ gear-rpm --verbose --commit -ba
```

При необходимости можно поправить `спес-файл` и добавить нужные зависимости. Когда пакет собирается и работа с ним закончена, стоит зафиксировать все изменения:

```
$ gear-commit -a
```

Команда оформит изменения и предложит их в утверждённом сообществом формате.

Обобщим опыт, упрощающий работу с `git`-репозиториями, предназначенными для хранения исходного кода пакетов.

- **Одна кодовая база — один репозиторий.**

Не имеет смысла хранить в одном репозитории исходные данные не связанных между собой пакетов. В этом правиле бывают исключения в случае, если культура разработки предполагает хранение разных составных частей проекта в одном репозитории.

- **Храните в `git`-репозитории распакованный исходный код.**

Исходные данные, поставляемые в виде архивов (`tar`, `zip`) и сжатые различными алгоритмами (`gz`, `bzip2`, `xz`), удобнее распаковать. Это упрощает использование средств `git` для работы с этими данными: так легче отслеживать изменения, ниже трафик при обновлениях и т. д.

- **Отделяйте свои изменения от изменений исходной кодовой базы.**

Если вы не разработчик пакета, который собираете, лучше хранить свои изменения отдельно от кода разработчиков, например в отдельной ветке (и формировать `diff` средствами `GEAR`) или в виде патчей. Это заметно упрощает совместную работу над пакетом, обновления и аудит.

6.5 Вопросы для самопроверки

1. Что такое инструмент `git`?
2. Что такое инструмент `GEAR`?
3. Где хранятся правила экспорта для `GEAR`?
4. Перечислите правила, упрощающие работу с `git`-репозиториями?
5. Какой формат у файла с правилами экспорта для `GEAR`?
6. Что означает параметр `@name@` в файле `.gear-rules`?
7. Как формируется путь в аргументах директив в файле `.gear-rules`?
8. Перечислите основные директивы в файле `.gear-rules`?
9. Какой командой собирается пакет с помощью `GEAR`?
10. Как создать тег, соответствующий версии пакета с помощью `GEAR`?

Глава 7

Примеры и упражнения

В главе приведены примеры для иллюстрирования работы инструментов по сборке программных пакетов. Примеры подготовлены для отечественной архитектуры микропроцессоров e2k (Эльбрус) и подходят для других процессорных архитектур. Различия состоят в путях, где упоминается имя используемой архитектуры.

7.1 Базовые инструменты rpm

Подготовка RPM-окружения

Рассмотрим пример сборки пакета средствами `rpmbuild` и использование некоторых команд утилиты `rpm`. Выполните следующие команды для настройки RPM-окружения:

```
$ sudo apt-get update
$ sudo apt-get install rpmdevtools rpm-build gcc-c++ git-core
$ rpmdev-setuptree
```



Проверьте, присутствует ли в системе директория `RPM`, введя команду: `$ ls ~/`

Если директория не присутствует в системе, то введите:

```
$ mv ~/rpmbuild ~/RPM
```

Откройте файл `~/ .rpmmacros`:

```
$ nano ~/ .rpmmacros
```

Если в файле есть другие строки, то удалите их. При заполнении файла применяйте только латинские символы. Содержимое файла должно иметь вид:

```
%_topdir           %homedir/RPM
%packager          Your_name Your_lastname <name@mail.domain>
```


Пример № 1. Сборка с rpmbuild

Скопируем необходимую версию примера проекта и перейдем в рабочий каталог. После символа обратного слеша «\» необходимо нажать Enter и продолжить ввод. Для однозначности точно укажем расположение рабочего каталога `/ExampleFirstProject/` в команде `git`:

```
$ git clone --branch=rpmbuild-v1 \
  https://gitlab.basealt.space/alt/edu/ExampleFirstProject.git \
  ~/ExampleFirstProject/
$ cd ~/ExampleFirstProject/
```

Сформируем `.tar`-архив с исходными данными проекта. Имя архива должно соответствовать инструкциям в `.spec`-файле:

```
$ tar cvf HelloUniverse-1.0.tar HelloUniverse
```

Расположим архив и `.spec`-файл в соответствии с правилами структуры каталогов RPM:

```
$ cp HelloUniverse-1.0.tar ~/RPM/SOURCES/
$ cp HelloUniverse.spec ~/RPM/SPECS/
```

Запустим сборку пакета с ключом `-ba` и передадим инструкции для сборки в структуре каталогов RPM.

```
$ rpmbuild -ba ~/RPM/SPECS/HelloUniverse.spec
```

В ходе выполнения команда скомпилирует исходные данные, соберет бинарный `.rpm`-пакет и создаст `src.rpm`-пакет с исходными данными.

Проверим наличие и внутреннюю информацию собранного бинарного пакета:

```
$ rpm -qpi ~/RPM/RPMS/e2kv5/HelloUniverse-1.0-alt1.e2kv5.rpm
```

Убедившись в наличии пакета, установим его в систему:

```
$ sudo rpm -i ~/RPM/RPMS/e2kv5/HelloUniverse-1.0-alt1.e2kv5.rpm
```

Теперь можно запустить исполняемый файл пакета и убедиться в его работоспособности:

```
$ HelloUniverse
```

К установленному в систему пакету можно обращаться по имени, заданному в метаданных. Посмотрим информацию о пакете, список файлов пакета и удалим пакет из системы:

```
$ rpm -qi HelloUniverse
$ rpm -ql HelloUniverse
$ sudo rpm -e HelloUniverse
```


7.2 Сборка в Hasher

Подготовка сборочного окружения

Для начала работы необходимо установить пакет **hasher** и настроить работу с утилитой:

```
$ sudo apt-get install hasher
```

Для начала работы необходимо настроить утилиту:

- Проверить версию пакета **hasher-priv**:

```
$ rpm -qa hasher-priv
```

Если версия **hasher-priv** старше или равна 2.0, то запустить демона:

```
$ sudo systemctl enable --now hasher-privd.service
```

- Подготовить учётную запись для работы с **hasher** можно средствами встроенной утилиты:

```
$ sudo hasher-useradd user
```

- Перезапустить рабочую сессию пользователя **user**:

```
$ su - user
```

- Создать каталоги расположения сборочной среды и каталоги настроек:

```
$ mkdir ~/hasher  
$ mkdir ~/.hasher
```

- Добавить ваше имя (только латинские символы) и отключение излишних проверок в условиях локальной сборки в конфигурационный файл. Открыть файл `~/hasher/config`:

```
$ nano ~/.hasher/config
```

Написать в него строки:

```
packager="Your_name Your_lastname <name@mail.domain>"  
no_sisyphus_check="packager,buildhost,gpg,changelog"
```


- Проверить права доступа файлов источников для пакетов системы `/etc/apt/`. Хотя бы один источник должен быть доступен на чтение для группы `hashman`. Для назначения группы на стандартные файлы источников можно выполнить команду:
`$ sudo chgrp hashman /etc/apt/sources.list{,.d/*}`

Если всё сделано верно, мы сможем создать минимальное сборочное окружение, исполнив команду:

```
$ hsh --initroot-only --verbose
```

Пример № 2. Сборка в Hasher разными способами

Hasher позволяет собирать пакеты в изолированном окружении по переданным специально подготовленным архивам `pkg.tar` и пакетам исходных данных `.src.rpm`. Он также позволяет работать в ручном режиме. Продемонстрируем все перечисленные варианты работы с инструментом Hasher.

Пример № 2.1. Ручная сборка в изолированном окружении

Используем Пример №1 и соберем пакет в ручном режиме для знакомства с базовыми командами Hasher.

В сборочное окружение необходимые зависимости нужно установить в ручную:

```
$ hsh-install gcc-c++
```

Уже имеющийся архив и `.spec`-файл поместим в изолированное сборочное окружение через служебные каталоги `/hasher/.in`:

```
$ cd ~/ExampleFirstProject/  
$ cp HelloUniverse-1.0.tar ~/hasher/chroot/.in/  
$ cp HelloUniverse.spec ~/hasher/chroot/.in/
```

Войдем в сборочное окружение Hasher:

```
$ hsh-shell
```

В окружение Hasher изначально мы окажемся в каталоге `/.in`. Скопируем архив и `.spec`-файл в уже подготовленные каталоги RPM/:

```
$ cp HelloUniverse-1.0.tar /usr/src/RPM/SOURCES/  
$ cp HelloUniverse.spec /usr/src/RPM/SPECS/
```

Запустим сборку:

```
$ rpmbuild -bb /usr/src/RPM/SPECS/HelloUniverse.spec
```


Полученный результат будет размещен в сборочной структуре RPM в каталоге `/usr/src/RPM/RPMS/`. Выйти из сборочного окружения можно нажав комбинацию клавиш `Ctrl+d` или исполнив команду:

```
$ logout
```

Пример № 2.2. Сборка с использованием `.src.rpm`-файла

Пример предполагает наличие пакета с исходными данными `.src.rpm` и позволяет в автоматическом режиме запустить сборку в изолированном окружении Hasher.

Используем полученный в примере № 1 `.src.rpm`-файл и передадим на сборку:

```
$ hsh --verbose ~/RPM/SRPM/HelloUniverse-1.0-alt1.src.rpm
```

Hasher распакует пакет, установит все необходимые зависимости, исполнит сборочные инструкции, проведет проверки и положит результат в каталог `/hasher/repo/`. Собранный в примере пакет можно найти в каталоге:

```
$ ls ~/hasher/repo/e2kv5/RPMS.hasher/
```

Пример № 2.3. Сборка с использованием `pkg.tar`-архива

В примере создадим архив `pkg.tar` в упрощенном виде. Такой архив с более строгими параметрами создает инструмент GEAR. Этот архив можно передать на сборку в Hasher.

Используем сформированный архив с исходными данными и `.spec`-файл, созданные в примере № 1, и упакуем в архив с указанными параметрами.

Перейдем в рабочий каталог и запустим команду архивирования:

```
$ cd ~/ExampleFirstProject/
$ tar -c --file=pkg.tar --label=HelloUniverse.spec \
  HelloUniverse-1.0.tar HelloUniverse.spec
```

Использованные ключи команды `tar`:

- `-c` — создать архив;
- `--file=pkg.tar` — задать имя архива (шаблонное имя архива);
- `--label=HelloUniverse.spec` — метка архива (должна совпадать с именем `.spec`-файла).

Созданный архив можно передать на сборку в изолированное сборочное окружение Hasher:

```
$ hsh --verbose ./pkg.tar
```

Проверим наличие собранного пакета:

```
$ ls ~/hasher/repo/e2kv5/RPMS.hasher/
```


7.3 Сборка с GEAR

Пример № 3. Использование GEAR в связке с другими инструментами

Для начала работы необходимо установить пакет `gear`:

```
$ sudo apt-get install gear
```

Скопируем необходимую версию примера проекта и перейдем в рабочий каталог.

```
$ git clone --branch=gear-v2 \
  https://gitlab.basealt.space/alt/edu/ExampleFirstProject.git \
  ~/ExampleFirstProject-v2/
$ cd ~/ExampleFirstProject-v2/
```

Загруженный проект дополнен инструкциями `.gear/rules`. По данным инструкциям, GEAR сам подготовит исходные данные для сборки и сформирует архив.

```
$ gear pkg.tar --verbose
```

Продemonстрируем как GEAR работает в связке с инструментами `rpmbuild` и `Hasher`.

Пример № 3.1. Сборка с rpmbuild

Убедитесь, что сборочное окружение `rpmbuild` присутствует в системе:

```
$ ls ~/.rpmmacros
```

Если файла не существует, то подготовьте сборочное окружение как написано в подготовке RPM-окружения (страница 58).

Выполним сборку с использованием `gear-rpm` и ключом `-bb` для сборки только бинарного `.rpm`-пакета:

```
$ cd ~/ExampleFirstProject-v2/
$ gear-rpm -bb
```

GEAR сформирует архив с исходными данными, архив и `.spec`-файл разложит в соответствии с определенной структурой и выполнит команду сборки в необходимом формате. Собранный пакет попадет в каталог `~/RPM/RPMS/`. Отобразим информацию о только что собранном пакете:

```
$ rpm -qip ~/RPM/RPMS/e2kv5/HelloUniverse-2.0-alt1.e2kv5.rpm
```


Пример № 3.2. Сборка с Hasher

Настройте инструмент **Hasher**, как описано в подготовке окружения (страница 60).

Запустим сборку пакета с использованием **gear-hsh**. Передадим ключи вида **--verbose -- --verbose** для подробного вывода информации о процессе выполнения как для **gear**, так и для вложенного **Hasher** инструментов:

```
$ cd ~/ExampleFirstProject-v2/  
$ gear-hsh --verbose -- --verbose
```

GEAR подготовит исходные данные и передаст в необходимом формате на сборку в изолированное сборочное окружение **Hasher**. Собранный в примере пакет можно найти в каталоге:

```
$ ls ~/hasher/repo/e2kv5/RPMS.hasher
```

Пример № 3.3. Сборка по тегу релиза с использованием автогенерируемого патча

Продemonстрируем сборку пакета по метке, связанной с ключевыми изменениями исходных данных. В нашем примере возьмем метку **v2.0** и дополнительно внесем собственные изменения с применением механизма **GEAR** для авто генерации патча.

Получим подготовленные примеры инструкций **.spec**-файла и **.gear/rules**:

```
$ cd ~/ExampleFirstProject-v2/  
$ git pull origin 2.0-alt2
```

Изменения в инструкциях позволяют собрать пакет с содержимым, связанным с меткой **v2.0** в истории **git**-репозитория. Но мы дополнительно внесем изменения в исходные данные нашего проекта и применим их средствами автогенерируемого патча.

Измените в файле **HelloUniverse/HelloUniverse.cpp** слово **New** на **Elbrus**:

```
$ nano HelloUniverse/HelloUniverse.cpp
```

Для сборки с помощью **GEAR** и **git** необходимо указать имя пользователя (только латинские символы) и адрес электронной почты того, кто будет проводить сборку:

```
$ git config --global user.name <username>  
$ git config --global user.email <name@domain.com>
```

Нужные изменения подготовлены, но не зафиксированы в истории **git**. Несмотря на это, мы можем применить изменения для локальной сборки, передав **GEAR** ключ **--commit**. Соберем пакет в изолированном окружении с дополнительным ключом **--lazy-cleanup** для сохранения файлов после окончания сборки.

Запустим сборку с перечисленными ключами:

```
$ gear-hsh --commit --lazy-cleanup --verbose -- --verbose
```

После завершения сборки ознакомимся с автоматически созданным и примененным патчем:

```
$ cat ~/hasher/chroot/usr/src/RPM/SOURCES/HelloUniverse-2.0-alt2.patch
```

Установим собранный пакет пакетным менеджером APT:

```
$ sudo apt-get install \  
~/hasher/repo/e2kv5/RPMS.hasher/HelloUniverse-2.0-alt2.e2kv5.rpm
```

Запустим исполняемый файл пакета:

```
$ HelloUniverse
```

Можем удалить пакет, используя пакетный менеджер APT:

```
$ sudo apt-get remove HelloUniverse
```

7.4 Обновление ранее собранного пакета

Пример № 4. Использование GEAR для обновления пакета

В примере будет рассмотрена упрощенная процедура обновления пакета с использованием инструментов «Альт платформы».

Чтобы продолжить работу, необходимо добавить набор утилит GEAR для настройки и взаимодействия с удаленными репозиториями:

```
$ sudo apt-get install gear-remotes-utils rpm-utils
```

Пример упрощен и объединяет в себе внешний (upstream) источник и рабочий источник примера. Скопируем версию примера, подготовленную для обновления:

```
$ git clone --branch=2.0-alt2 \  
https://gitlab.basealt.space/alt/edu/ExampleFirstProject.git \  
~/ExampleFirstProject-v3/  
$ cd ~/ExampleFirstProject-v3/
```

Назначим наш базовый-рабочий источник (origin) внешним (upstream). Такой вариант используется для примера и в обычной рабочей ситуации подразумевает различие источников:

```
$ gear-remotes-save origin
```

Теперь возможно обращаться к нашему источнику примера как к **upstream**. Притянем новые изменения и проверим новые метки изменений:


```
$ git fetch upstream
$ gear-remotes-watch
```

Теперь притянем изменения из метки третьей версии в рабочее файловое дерево:

```
$ git merge v3.0
```

Обновим инструкции для сборки исходных данных с меткой `v3.0`:

```
$ gear-update-tag v3.0
```

Откройте `.spec`-файл в текстовом редакторе:

```
$ nano ~/ExampleFirstProject-v3/.gear/HelloUniverse.spec
```

Измените версию и релиз на следующее:

```
Version: 3.0
Release: alt1
```

Все остальные поля остаются неизменными.

Утилита `add_changelog` дополняет секцию журнала `.spec`-файла списком изменений в утвержденном формате (каждую строку списка принято начинать с символа `"-"`). Создадим запись в журнале для `HelloUniverse.spec`:

```
$ add_changelog -e "- Updated to new version v3.0." \
~/ExampleFirstProject-v3/.gear/HelloUniverse.spec
```

Добавим все изменения в историю `git` и сформируем комментарий к изменениям одной командой:

```
$ gear-commit -a --no-edit
```

Соберем пакет и проверим результат:

```
$ gear-hsh --verbose -- --verbose
```


Заключение

Уважаемый читатель!

В данном пособии вы познакомились с различными способами сборки программных пакетов с помощью утилит: `rpmbuild`, `Hasher`, `GEAR`. Научились собирать пакеты для отечественной архитектуры микропроцессоров Эльбрус — Е2К.

В качестве направления дальнейшего развития, попробуйте собрать свой дистрибутив. Для этого вам понадобится ещё один инструмент: `mkimage-profiles`¹. `Mkimage-profiles` — система управления конфигурацией семейств дистрибутивов «ALT» для различных платформ. Лицензионное соглашение на «Альт Платформа»² позволяет вам создавать свой дистрибутив под свободной лицензией GPL v3.

¹<https://www.altlinux.org/Mkimage-profiles>

²https://www.basealt.ru/fileadmin/docs/License_Alt-Platform_10.pdf